Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Jacek Sroka

Models and languages for specification of collection-oriented scientific workflows

PhD dissertation

Supervisors

dr hab. Jerzy Tyszkiewicz
and
dr Jan Hidders

Institute of Informatics
Warsaw University

October 2008

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

October 24, 2008 .....
date *Jacek Sroka*

Supervisor's declaration:
the dissertation is ready to be reviewed

October 24, 2008 .....
date *dr hab. Jerzy Tyszkiewicz*

October 24, 2008 .....
date *dr Jan Hidders*

Abstract

In many sciences, like ecology, geology, chemistry, astronomy and especially bioinformatics, structured data is analyzed with the use of collection-oriented scientific workflow (COSW) systems. Such systems allow to describe the experiments with a kind of network, through which the data flows and is processed, and where the nodes of the network carry out domain specific operations.

Many specialized COSW workbenches exist and are based on simple yet expressive graphical notations, integrate most important tools, services and databases from a given domain, and include various additional useful features like data provenance tracking or service discovery. The models, languages and techniques used in COSW modeling have become an interesting topic of study themselves and are the focus of this thesis. They have many relationships with workflow modeling, business process modeling, databases, computational grids and many more established research areas.

The main contributions of this thesis are: (1) investigation and formalization of the semantics of Scufl — the COSW specification language of a popular Taverna workbench, and (2) the creation of a new formal model for specification of COSWs, that combines both the control flow and data manipulation aspects and is as close as possible to the existing models from workflow and database domains allowing to reuse available theoretical results.

Key words

DFL, collection-oriented scientific workflow, workflow systems, Petri net, nested relational calculus

Thematic classification

D. Software
D.3 Programming languages
D.3.2 Language classifications
Data-flow Languages

Contents

Chapter 1

Introduction

1.1 Collection-oriented scientific workflows

Information technology techniques and results developed for business applications are constantly challenged by new needs emerging from dynamically growing applied sciences. This is especially true for the domains of database systems and workflow processing, since even larger volumes of data have to be analyzed and the analysis processes become even more complex. Where those two domains coincide a new interesting field of research on *collection-oriented scientific workflows* (COSWs) emerges.

In many sciences, like ecology, geology, chemistry, astronomy and especially bioinformatics, structured data is analyzed by a software system organized into a kind of network, through which the data flows and is processed, and where the nodes of the network carry out domain specific operations. This is similar to doing workflow processing in business, but here more emphasis is put on the processing of collections of data values and less on the control flow issues, hence we propose the term collection-oriented scientific workflow (COSW).

The basic operations in such workflows are mainly specialized, domain-specific data analysis algorithms. Their efficient implementations are available as open source tools or are made freely accessible on dedicated Internet servers maintained by scientific institutions. The results produced by the workflows are used to form scientific hypotheses and to justify or invalidate them. The way a COSW is organized, i.e., which operations are executed and how they depend on each others results, is important and is usually

published in some form together with the results of the data processing experiment. This is necessary for the reviewers and readers to understand what was done in the experiment, to effectively and objectively assess its merit, to repeat and verify it, and finally to adapt it for their own research projects.

Traditionally such data processing experiments, have been performed by copying and pasting data between local programs, e.g., the components of the EMBOSS package [52], and web accessible processing servers with WWW forms type user interfaces, like FASTA Sequence Comparison at the University of Virginia [61] and the Basic Local Alignment Search Tool at NCBI [42]. This method of experimenting is laborious and error prone. It has also been common to construct *ad hoc* scripts and programs to automatize this task, but for that at least some basic knowledge of programming and distributed programming issues is necessary. Furthermore, the produced software usually has been not portable and poorly, if at all, documented.

Nowadays, specialized scientific workflow workbenches such as Taverna [45, 30] and Kepler [35] are used. They are based on simple yet expressive graphical notations, integrate most important tools, services and databases from a given domain, and include various additional useful features like data provenance tracking or service discovery. The models, languages and techniques used in COSW modeling have become an interesting topic of study themselves and are the focus of this thesis. They have many relationships with workflow modeling, business process modeling, databases, computational grids and many more established research areas.

1.2 Existing systems

In this section we list popular existing scientific workflow systems. We also review three that seem to be the most interesting in the context of this thesis, i.e., widely used Taverna, incorporating multiple models of computation Kepler and BioKleisli which has a strong theoretical background. This choice has been mainly motivated by the interesting design of models and languages for specifying workflows that those systems are based on.

All three systems come from the domain of bioinformatics, which is presently a thriving research area. The application of scientific workflow systems in bioinformatics is a sign of strife to transfer the emphasis of the biology research from wet laboratory to computer laboratory, i.e., to conduct as large part of the experiments as possible *in silico* as opposed to

traditionally doing everything *in vitro*. If accomplished this allows to test quickly and cheaply scientific hypotheses before engaging in time consuming wet laboratory tests, i.e., with the use of test tubes and expensive chemical reagents.

1.2.1 Taverna workbench

Taverna [45, 30] is an easy to operate workbench for COSW development and enactment. It allows users to graphically construct COSWs from libraries of available components and is intended for use in bioinformatics data analysis experiments. The most important virtues of Taverna are that it is very easy to use, has a specialized and expressive graphical specification language and integrates thousands [46] of data analysis tools. It also includes additional useful features like service discovery, storing intermediate results and tracking data provenance. The workbench is being constantly developed, but it is already considered stable and has been used in real life research, e.g., [57, 34].

A small example of a Taverna COSW is given in Fig. 1.1 (a). A set of workflow inputs is indicated by a dotted rectangle with a small triangle pointing upwards, which in this case contains one input labeled *pin*. The graph also contains a set of workflow outputs indicated by a dotted rectangle with a small triangle pointing downwards, here containing two outputs labeled *ppout* and *pout*. Furthermore, the graph contains several so-called processors which represent operations from the Taverna services and which are labeled "Get_Nucleotide_FASTA", "Merge_String_list_to_string", "emma", "showalign" and "prettyplot". For each processor, depending on the view settings, the input ports are listed in the top row, as is done here, or in the left column, as in some of the following examples. Similarly, the output ports are listed in the bottom row or in the right column. For example, the processor with label "emma" has one input port labeled *sequence_data_direct* and one output port labeled *outseq*.

The COSW defines a simple yet often needed experiment. If a *pin* input port is initiated with a list of nucleotide sequence identifiers, then the "Get_Nucleotide_FASTA" processor implicitly iterates on this list and with the use of an external service that searches the GenBank database [6] returns FASTA formatted nucleotide sequences that correspond to the identifiers. The next processor merges the list of those sequences into one long string, on which the "emma" processor, which is a wrapper for the ClustalW operation of the EMBOSS [52] package, performs a sequence alignment. The final

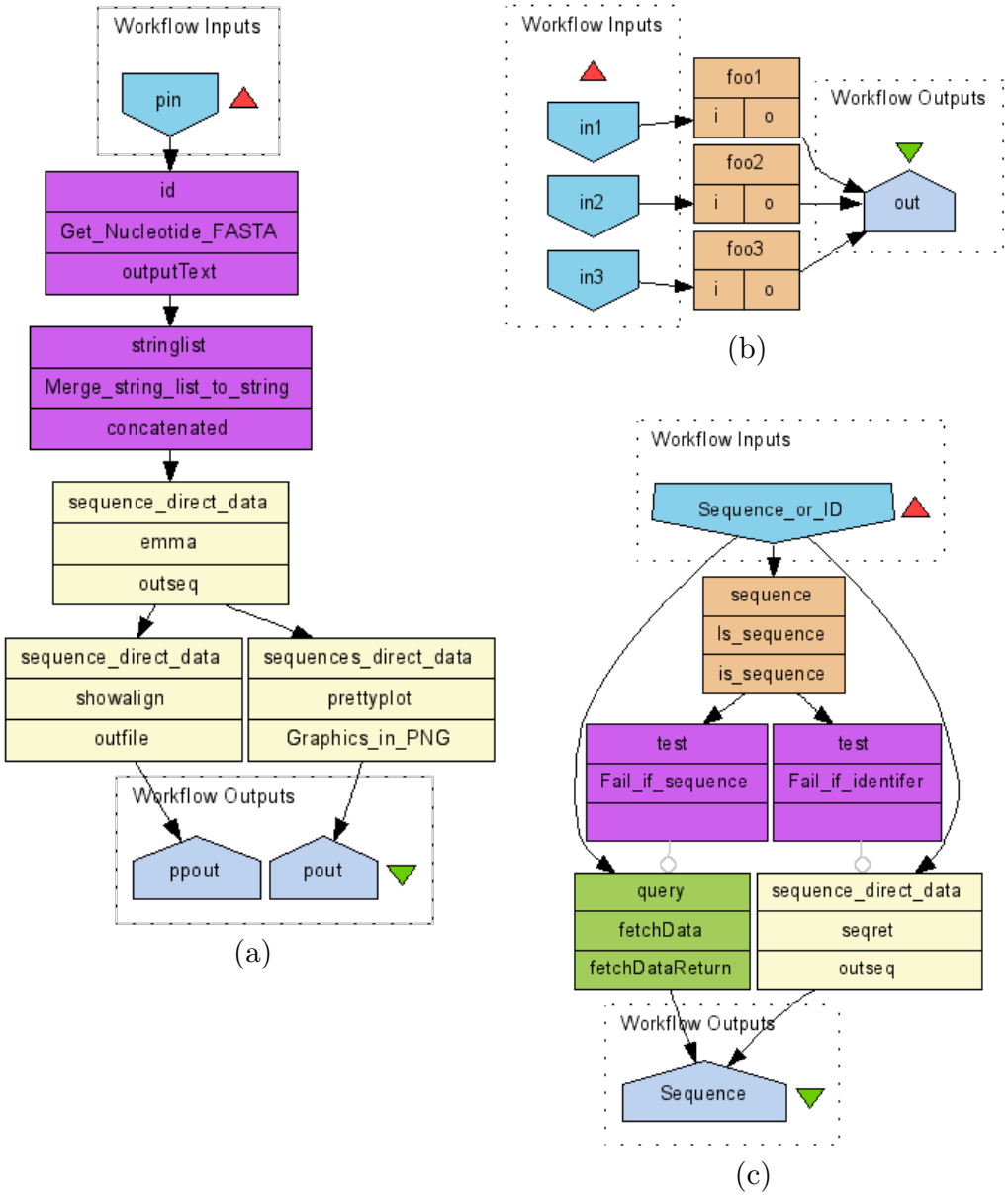

Figure 1.1: Examples of COSWs in Taverna

two processors, "showalign" and "prettyplot" are used to present the output respectively in a textual and graphical manner.

As can be noticed, the graphical representation of the COSW communicates well the main intent of the experiment. In the following examples we introduce other important features of the system and then we proceed with formal definitions and discussions of these features.

The second example is abstract and is presented in Fig. 1.1 (b). The graph has three branches that independently process their own input values. All the computed values, i.e., the results of "foo1", "foo2" and "foo3" processors, are directed to the *out* workflow output. Although it is not visible in the graphical representation of the COSW, for the *out* workflow output an *incoming-links strategy* is specified. It determines how the value for a port is obtained in case of multiple data edges ending in it. This strategy can be either *merge* or *select-first*, where merge waits for values to arrive from all incoming data edges and packs them into a list while select-first selects the first value that arrives and ignores the others. Example use cases for the different incoming-links strategies in this abstract COSW would be:

- for the merge strategy — obtaining nucleotide sequences from a number of databases and packing them together into a list for further processing, e.g., alignment,
- for select-first strategy — requesting the same computation with different services and continuing the processing with the result that arrives the quickest.

An extra feature of the select-first strategy is that the COSW is less error prone. In Taverna processors can fail, for example if no connection can be made with them over the Internet. Here the COSW finishes properly if at least one of the used tools, i.e., "foo1", "foo2" or "foo3", finishes with success.

The third example is taken from the myExperiment [23] workflow repository. It is presented in Fig. 1.1 (c) and is incorporated as a building block into several COSWs defining real-life *in silico* experiments that are also published in the repository. First, it shows that in the Taverna COSWs there are two kinds of edges. The *data edges*, indicated by solid edges with an arrow head, represent data flow by connecting workflow inputs or output ports of processors with input ports of processors or workflow outputs. The *control edges* indicated by gray edges ending with a circle represent additional control flow. They connect two processors specifying

that one can execute only when the other has successfully finished. Second, the example presents how a combination of failing processors, control edges and ports with many incoming edges and the select-first strategy specified can be used to model conditional behavior. The COSW returns a sequence in a FASTA format that corresponds to a sequence or sequence entry identifier provided as an input. If a sequence identifier, in `database:identifier` format, e.g. `uniprot:wap_rat`, is provided as the input, then the "Fail_if_sequence" processor succeeds but the "Fail_if_identifier" fails and thus the "fetchData" processor uses the EBI's WSDbfetch web service (see `http://www.ebi.ac.uk/Tools/webservices/services/dbfetch`) to retrieve the sequence in FASTA format. Otherwise the "Fail_if_sequence" processor fails but the "Fail_if_identifier" succeeds and the sequence is passed through the Soaplab [33, 53] "seqret" service to force it into a FASTA format. Both conditional branches are joined with the *Sequence* workflow output for which the select-first strategy is specified.

Taverna includes other interesting features like product strategies and nested COSWs, which are beyond this short presentation. A further and complete discussion of this system is given in Chapter 2.

1.2.2 Kepler

The Kepler scientific workflow system [35] is applied in many areas including bioinformatics, ecology, oceanography and geology. It is based on the Ptolemy II [17] system, which is a modeling and simulation environment, and extends it with new features and components for COSW design and for efficient workflow execution. Similarly as in Taverna (see 1.2.1) COSWs are specified as graphs where the nodes, here called *actors*, have input and output ports and represent scientific operations. Ports can be connected with *channels* to define the flow of data. Additional control flow constructs, as loops and branches, are also present and COSWs can be nested with the use of *nested actors*. An example COSW with a legend is presented in Figure 1.2. It is an adapted version of an example provided with the Kepler user documentation. It solves a discrete finite-difference equation, that determines a resource-limited population growth, with a growth factor "r", and a carrying capacity "k".

A distinguishing feature of Kepler is its ability to enact one and the same COSW according to different computation models which the COSW author specifies with the so called *director*. Kepler includes directors that correspond

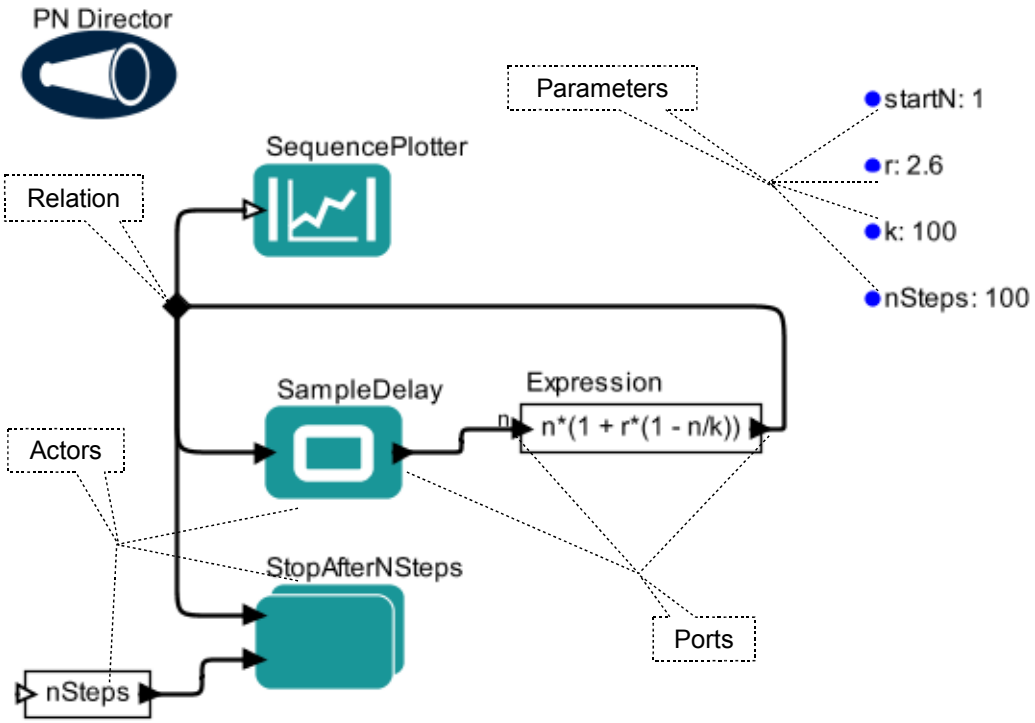

Figure 1.2: An example of a COSW in Kepler

to process network, synchronous dataflow, continuous time, discrete event, and finite state machine computation models. In our example, both the top level COSW and the nested COSW *StopAtEndN*, use the process network director. Together with its restriction — the synchronous dataflow director — they are the most frequently used ones. We will characterize the process network model by explaining how the example from Fig. 1.2 works.

The execution in process network model is driven by input data availability, i.e., an actor can fire, if some input data tokens are available on all its input ports. On the output ports it produces tokens with the result values, which are immediately transfered to further actors. Yet, there are exceptions. In the example from Fig. 1.2 a special actor *SampleDelay* is used to start the loop. Without needing any input data tokens on its single input port, it fires once and produces a token with the value assigned to COSW parameter *startN*. Later on it behaves as an identity operation, which outputs the value it consumes. The value produced by the *SampleDelay* actor

is used as the previous population size "n" by the actor that evaluates the formula. Then, three copies of the newly computed population size are made. The first is provided to the *SequencePlotter* actor that updates a diagram with the computed results. The second is provided to the *SampleDelay* actor to continue the loop. Finally, the third is provided to a nested COSW *StopAfterNSteps*, which contains a *Counter* actor that has an internal state and counts the number of times it is executed. When that value reaches the parameter *nSteps*, which is endlessly provided to the nested COSW by a *Constant* actor *nSteps*, the *StopAfterNSteps* nested COSW uses another actor it contains, namely *Stop* actor, to finish the computation. If a *Stop* actor was not used explicitly, then the COSW would continue indefinitely since the *nSteps* constant actor would not stop producing tokens and the loop would continue.

In Kepler polymorphic COSWs can be defined, as discussed in [37]. The definition of polymorphic COSWs is based on a technique that is similar to that of the implicit iteration in Taverna, where an actor that expects input of a certain type can also operate on collections of other types by automatically identifying nested values of the right type and operating on them. This type of polymorphic actor is referred to by the authors of [37] as *collection-aware actors*. A difference with Taverna is that the user can specify in more detail how such nested values are identified and how they are iterated over, where in Taverna this is completely transparent. Another difference with Taverna is that Kepler features an elaborate and refined type system which explicitly allows heterogeneous values and this type system is used in the specification of the aforementioned collection-aware actors.

Thanks to the inclusion of different computational models that can be combined in one COSW, the availability of loops and the presence of special actors like the *Counter* actor, which has an internal state, or the *SampleDelay* actor, which does not need input tokens to fire for the first time, Kepler is very expressive as compared to other COSW systems. By expressive we mean here that difficult problems can be solved in Kepler with COSWs of small size. The trade off is that Kepler COSWs are hard to analyze with formal methods and difficult to understand by users with small programming experience.

1.2.3 BioKleisli

BioKleisli [14] is a system allowing querying and transforming complex data from heterogeneous sources, including ones available through a network,

which was developed for and used in the Human Genome Project [62]. It is not a COSW system *per se*. It provides neither graphical notation nor deals with control flow issues, but is very interesting because of the way how complex collections of data can be processed and because of its theoretical background. It is based on a query language called Collection Programing Language (CPL) [71], which is amenable to optimizations and which on nested relational data has the expressive power of nested relational algebra [70]. The design of CPL is based on nested relational calculus (NRC) [9] (see 3.2.1) — a calculus version of the nested relational algebra, which is a well studied formalism for querying nested data collections and for which many optimization results are available. It is worth pointing out that in separate research [21, 22] we also studied the usefulness of NRC for COSW specification and analysis. NRC and CPL follow a new approach to query languages inspired by *structural recursion* [58] and the category theory notion of a *monad* [68, 39]. They are also *type orthogonal* [14], which means that the design of the language is structured around its type system.

The type system of CPL is given by:

$$\begin{aligned}\tau \ ::= \ & \mathit{bool} \mid \mathit{int} \mid \mathit{string} \mid \ldots \mid \{\tau\} \mid \{\!|\tau|\!\} \mid \{\!\|\tau\|\!\} \mid \\ & [l_1 : \tau_1, \ldots, l_n : \tau_n] \mid \ \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\end{aligned}$$

where *bool*, *int*, *string*, ..., are the base types, $\{\tau\}$, $\{\!|\tau|\!\}$ and $\{\!\|\tau\|\!\}$ are respectively set, bag and list types from the type τ, and $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ and $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$ are respectively record and variant record types with element types $\tau_1, \ldots, \tau_n$ and field labels $l_1, \ldots, l_n$. The semantics of those types is defined as usual.

The operations of CPL are based on NRC yet the syntax follows Wadler's work [68] in order to make it more user friendly. CPL is also more robust, for example it is equipped with *pattern matching* and allows for function definition within the language. We will not give here a complete definition, but limit ourselves to a few simple queries on a database containing a set of COSW system descriptions of type:

$$\begin{aligned}\mathit{Systems} = \{[&\mathit{name} : \mathit{string}, \\ &\mathit{domains} : \{\mathit{string}\}, \\ &\mathit{authors} : \{\mathit{string}\}]\}\end{aligned}$$

An example fragment of data conforming to this type is:

$$DB = \{[name = \text{“BioKleisli”},\\ domains = \{\text{“bioinformatics”}\},\\ authors = \{\text{“S. Davidson”}, \text{“C. Overton”}, \text{“V. Tannen”}, \text{“L. Wong”}\}\\ ], \ldots\}$$

In the query:

$$\{[name = s.name, domains = s.domains] \mid \backslash s <- DB\}$$

on the right-hand side of $\mid$ the variable $\backslash s$ traverses the set DB. The result set is constructed on the left-hand side of $\mid$ by projecting the records to fields *name* and *domains*. In the case of bag construction the duplicates would be kept and for lists the order of traversal would be maintained.

In the next query:

$$\{[domains = d] \mid [name = \text{“BioKleisli”}, domains = \backslash d, \ldots] <- DB\}$$

only the records with *name* = “BioKleisli” contribute to the result. As we can see the ellipsis “. . . ” can be used to match the remaining record fields.

A normalization of the data on names and domains can be achieved with the query:

$$\{[name = n, domain = d]\\ \mid [name = \backslash n, domains = \backslash dd, \ldots] <- DB, \backslash d <- dd\}$$

which results in a set of string pairs with a COSW system name and one of its domains. Note, that this and the next query would not be possible in the flat relational calculus.

Finally, the query:

$$\{[domain = d, names = \{x.name \mid \backslash x <- DB, d <- x.domains\}]\\ \mid \backslash y <- DB, \backslash d <- y.domains\}$$

restructures the records so that for each domain a set of COSW system names is stored as opposed to a set of domains for each COSW system name.

1.2.4 Other systems

Some other COSW systems include:

- DiscoveryNet [54] — from the domains of bioinformatics and chemistry;
- Triana [36] — from the domain of astronomy;
- Pegasus [15] — from the domains of astronomy and bioinformatics;
- SCIRun [32] — general usage, applied in biomedicine and bioelectric field modeling;
- JOpera [48] — general usage web service composition tool, applied in bioinformatics.

The newest systems can be tracked with help of on-line survey sites as [55, 24], that are maintained by the scientific community. Also business workflow tools can be used to define and enact scientific workflows, yet they are usually control flow oriented, i.e., lack the ability to manipulate complex collections of data, and an additional work is required to integrate services from scientific domains.

1.3 Problems that will be addressed

We start with studying and formalizing the semantics of the process model of the Taverna workbench, which is one of the most popular COSW systems used in bioinformatics. Our goal is to precisely and comprehensively describe all its features, especially those that distinguish it from other workflow systems, examine their usefulness and discuss alternatives. Although we strive for an elegant formal model, we make only a minimal number of compromises in order to describe the current Taverna implementation as faithfully as possible. This way when later a clean core model of Taverna is devised it is possible to asses its fidelity.

Because the formalization of Taverna turns out to be very involved, the question rises whether it is possible to design a simpler language with an easier to understand formal semantics that can describe COSWs. For this purpose we investigate if and how existing results on databases and classical workflow modeling can be applied in the COSW research. There exist simple,

clean and well studied formal models of workflows which deal only with control flow and ignore processing of nested collections of data, e.g., Petri nets (see 3.2.2). Similarly, simple, clean and well studied formal models exist for dealing with nested collections of data, but ignore the specification of control flow, e.g., NRC (see 3.2.1). In the remainder of the thesis we are concerned with the creation of a new hybrid formal model for specification of COSWs from first principles, that combines both the control flow and data manipulation aspects. Our aim is to structure the new model as close as possible to the existing models from both domains, such that the reuse of results available for them is possible. We also study if and how such a hybrid formal model can be useful in practice, i.e., in real life scientific workflow experiments. For that we construct a new COSW system and test it on real-life experiments adapted from Taverna workbench.

1.4 Structure of the thesis

In Chapter 2 we have investigated and formalized the semantics of Scufl — the COSW specification language of Taverna workbench. Then we use the semantics to prove some basic properties of all COSWs defined in Scufl.

In Chapter 3 we formally define DFL (as in **D**ata**F**low **L**anguage) — a new language for specifying COSWs which is a combination of Petri nets and NRC. In Section 3.7 we present that certain results for its components can also be applied for DFL. Then, in Section 3.8, we present a tool that allows one to design, enact and analyze DFL COSWs.

Finally, in Chapter 4, we summarize our results and indicate interesting areas and problems for further research that are motivated by the results covered in this thesis.

Chapter 2

Scufl

In this chapter we investigate and formalize the semantics of Scufl, study its distinguishing features, examine their usefulness and discuss alternatives. The formal definitions which we give not only allow to precisely understand what is really being done in a given experiment. They are also the first step toward automatic correctness verification and allow the creation of auxiliary tools that would detect potential errors and suggest possible solutions to COSW creators, the same way as Integrated Development Environments aid modern programmers. The creation of formal semantics is also essential for work on enactment optimization and in designing the means to effectively query COSW repositories.

2.1 Motivation of formal semantics for Scufl

Scufl includes high level features and mechanisms, like implicit iteration, that make the construction of real life COSWs simpler and allow the programmer to focus on the problem being solved. At the same time the COSWs look less complex and can be used in research papers to convey the main idea of an *in silico* experiment that was conducted. Yet, distributed data-processing experiments are complex in nature and a highly expressive definition language that hides much of the complexity of the COSW behind implicit semantics is not the silver bullet. When problems appear, e.g., while debugging, it is important to exactly understand what computation is being done. And even when the specification of the COSW is successfully finished, it's merit has to be effectively and objectively assessed by reviewers. For this a precise and

formal semantics is needed.

It's also obvious that the *in silico* experiments that are being conducted become more and more complex and sooner or later automatic verification procedures, similar to those used for verifying complex business transactions, will have to be developed. For such verification the existence of formal semantics is a necessary first step as well as for the creation of auxiliary tools that would detect potential errors and suggest possible solutions to COSW creators, the same way as Integrated Development Environments aid modern programmers.

Another domain for which a formal semantics is fundamental is enactment optimization. As with database queries the programmer could only specify what has to be done and the determination of the most effective execution strategy would be left to the COSW engine. In addition, with COSWs being applied more and more frequently, and being shared in Internet repositories [23], their querying is becoming an interesting scientific problem [10, 5]. A successful COSW query language should take into account the semantics and not just the syntax, i.e., compare what the COSWs do and not only how they are defined.

Finally, we argue that the very act of formulating a formal semantics is useful because it forces us to do a complete and thorough analysis of the behavior of Taverna. The formulation of an elegant and natural formal semantics is a good litmus test for checking if the current behavior is consistent and well chosen. Such a test is not unimportant for large, complex and relatively rapidly evolving systems such as Taverna. In addition, as is shown in this thesis later on, it may provide inspiration for other interesting alternative semantics. Therefore the formulation of a formal semantics can help in the future design and development of Taverna.

2.2 Scufl type system

As the Taverna authors notice "the problem of data typing in life sciences is simply too hard to attack". There is only one basic type that describes binary data with an attached MIME annotation and we will denote this basic type as $\mathcal{M}$. The MIME annotation is used to determine how a basic type data value is going to be presented to the user, e.g., whether a text, a picture, or its binary representation is going to be displayed. The set of MIME values is denoted as $\mathcal{V}_{\mathcal{M}}$. For our examples we will usually assume it contains at least

the natural numbers and strings.

In Taverna we meet in practice only one collection type, namely, ordered lists, even though the documentation suggests that Scufl was designed to support other collection types such as partial orders, trees, bags and sets. Although the user documentation mentions only homogeneous lists, the workbench does not prevent the use of heterogeneous lists, i.e., lists containing elements of different types such as $[1, [2], 3, [[4]]]$. Heterogeneous lists can be obtained from homogeneous ones during the computation. For example, it is possible to specify in a Taverna COSW that an input is computed from different outputs of different processors by combining them into a single list. Therefore we define the set of complex values such that it includes heterogeneous lists.

The *set of complex values*, denoted as $\mathcal{V}_{tav}$, is defined as the smallest set such that (1) $\mathcal{V}_{\mathcal{M}} \subseteq \mathcal{V}_{tav}$ and (2) if $x_1, \ldots, x_n \in \mathcal{V}_{tav}$, then the list $[x_1, \ldots, x_n]$ is in $\mathcal{V}_{tav}$. The values of these list types will be denoted as $[1, 2, 3]$ and $[[1, 2], [3, 4], 5]$, the empty list is denoted as $[]$, and the concatenation of lists is denoted with $+$, so $[1, 2] + [1, 5] + [] = [1, 2, 1, 5]$. Note, that this notion of complex value does not include tuples or records.

Although heterogeneous lists can appear in Taverna, they usually cause processors to fail and otherwise are not always processed coherently, e.g., applying the flatten operation to the list $[[x], [[y]]]$, where x and y are some basic values, results in $[[x], [y]]$ while flattening of $[[[x]], [y]]$ results in $[[x], y]$. It is however quite possible to give an intuitive semantics for Scufl that allows heterogeneous values everywhere and deals with them consistently. Therefore, we will in the formal part of this chapter, for the sake of simplicity and consistency, assume that heterogeneous values are allowed everywhere. If heterogeneous values never appear, then the semantics defined in this chapter corresponds to the observed behavior of Taverna.

The consistent behavior for the heterogeneous values is owed to the coherent generalization of semantics of product strategies expressions (see Section 2.5.4) and implicit iteration mechanism (see Section 2.5.2). Despite this we usually limit the presentation to homogeneous values only and discuss in Section 2.7 the strategies for adapting the semantics such that the heterogeneous values are consistently avoided.

Although Taverna does as little typing as possible it still has a notion of *complex type*, which is defined by the following syntax:

$$\tau \quad ::= \quad \mathcal{M} \mid [\tau]$$

Examples of such types are $\mathcal{M}$, $[\mathcal{M}]$, $[[\mathcal{M}]]$, *et cetera*. The set of all complex types is denoted as $\mathcal{T}_{tav}$. The semantics of these types are defined with induction on their syntactic structure such that:

- $[\![\mathcal{M}]\!] = \mathcal{V}_{\mathcal{M}}$, and
- $[\![[\tau]]\!] = [\![\tau]\!] \cup \mathcal{L}([\![\tau]\!])$ where $\mathcal{L}(V)$ denotes the set of finite lists over V.

Note, that the given type semantics is more liberal than usual and explicitly allow heterogeneous lists. So not only $[[1],[2]] \in [\![[[\mathcal{M}]]]\!]$ but also $[1,[2]] \in [\![[[\mathcal{M}]]]\!]$ since $1 \in [\![\mathcal{M}]\!] \Rightarrow 1 \in [\![[\mathcal{M}]]\!]$. Effectively the type only restricts the maximum nesting depth of the complex values in its semantics.

Further motivation for the liberal list type semantics is given by the fact that if the nesting depth of a certain value is lower than expected there is always an intuitive interpretation of that value as a more deeply nested one, namely by nesting it in singleton lists. For example, if a certain processor expects on a certain input port a list of protein identifiers and it receives a value that is an unnested single protein identifier, then it can interpret this as a singleton list containing this protein. This principle can be applied to every type, i.e., a value of type τ can always be interpreted as a value of type $[\tau]$ by assuming it is packed in a singleton list. This is reflected in the type semantics by the fact that $[\![\tau]\!] \subseteq [\![[\tau]]\!]$. The idea that types are given a semantics that is related to a coercion mechanism can be found in other work such as [4].

Consistently with the given type semantics and the described type coercion we define a subtyping relation, denoted by $\sqsubseteq$, over complex types such that $\tau \sqsubseteq \sigma$ iff the nesting depth of τ is less than or equal to the nesting depth of σ, i.e., either $\tau = \mathcal{M}$, or $\tau = [\tau']$ and $\sigma = [\sigma']$, where $\tau' \sqsubseteq \sigma'$. For example, $\mathcal{M} \sqsubseteq [\mathcal{M}]$, and $[\mathcal{M}] \sqsubseteq [[[\mathcal{M}]]]$, but $[[\mathcal{M}]] \not\sqsubseteq [\mathcal{M}]$. Clearly, this notion of subtyping is consistent with the given semantics, i.e., for all complex types τ and σ it holds that $\tau \sqsubseteq \sigma$ iff $[\![\tau]\!] \subseteq [\![\sigma]\!]$.

Since there is only one basic type, viz. $\mathcal{M}$, it is not hard to see that $\sqsubseteq$ defines a linear order over the complex types. So we can define a function $\mathsf{max} : \mathcal{P}(\mathcal{T}_{tav}) \rightarrow \mathcal{T}_{tav}$, such that $\mathsf{max}(T)$ is the least common upper bound of T, i.e., the smallest complex type σ such that for all types $\tau \in T$ it holds that $\tau \sqsubseteq \sigma$. This means, for example, that $\mathsf{max}(\emptyset) = \mathcal{M}$, $\mathsf{max}(\{\mathcal{M}, [\mathcal{M}]\}) = [\mathcal{M}]$, and $\mathsf{max}(\{[\mathcal{M}], [[[\mathcal{M}]]]\}) = [[[\mathcal{M}]]]$.

2.3 Scufl global components

Here we list the Scufl components that are common to all COSW. We postulate a countably infinite set PL of *port labels* that contains all names that can be given to input and output ports of processors as well as to workflow inputs and outputs. The Taverna workbench comes with a huge library of built-in bioinformatics operations, which are mainly external service intermediaries, i.e., programs that call external services. We call this extensible collection of operations the *Taverna services* and model it by a set of service names called TS which can contain an arbitrary number of names.

The interface of a service is defined by tuple types that give the input type and the output type. These tuple types are defined as partial functions $\sigma : PL \to \mathcal{T}_{tav}$ that map a finite subset $\mathrm{dom}(\sigma) \subseteq PL$, called the *domain* of σ, to complex types. We will denote tuple types $\{(l_1, \tau_1), \ldots, (l_n, \tau_n)\}$ as $\langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$. The set of all tuple types is denoted as $\mathcal{T}_{tup}$ and the set of all tuple values as $\mathcal{V}_{tup}$. The semantics of a tuple type $\sigma = \langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$, denoted as $[\![\sigma]\!]$, is defined as the set all functions $t : \mathrm{dom}(\sigma) \to \mathcal{V}_{tav}$ such that for each $l_i \in \mathrm{dom}(\sigma)$ it holds that $t(l_i) \in [\![\tau_i]\!]$. Such a function $\{(l_1, x_1), \ldots, (l_n, x_n)\}$ will be denoted as $\langle\, l_1 = x_1, \ldots, l_n = x_n \,\rangle$. For later use we define a notation for the projection of a tuple type σ on a set of labels L as $\sigma|_L$ such that $\sigma|_L = \{(l, \tau) \in \sigma \mid l \in L\}$ and its counterpart for tuple values as $t|_L = \{(l, v) \in t \mid l \in L\}$.

To define the interface of the Taverna services we postulate the functions $type_i : TS \to \mathcal{T}_{tup}$ and $type_o : TS \to \mathcal{T}_{tup}$ that give the input type and output type, respectively, of each service as a tuple type. In addition we define the functions $I : TS \to \mathcal{P}(PL)$ and $O : TS \to \mathcal{P}(PL)$ such that for every service name $s \in TS$ $I(s)$ gives the set of input port labels and $O(s)$ the set of output port labels, i.e., $I(s) = \mathrm{dom}(type_i(s))$ and $O(s) = \mathrm{dom}(type_o(s))$. For example, the interface for the string concatenation operation "Concatenate_two_strings" $\in TS$ is defined as follows (we abbreviate "Concatenate_two_strings" to "c_t_s"):

$$\begin{aligned} I(\text{“c\_t\_s”}) &= \{string1, string2\} \\ type_i(\text{“c\_t\_s”}) &= \langle\, string1 : \mathcal{M}, string2 : \mathcal{M} \,\rangle \\ O(\text{“c\_t\_s”}) &= \{output\} \\ type_o(\text{“c\_t\_s”}) &= \langle\, output : \mathcal{M} \,\rangle \end{aligned}$$

The semantics of a service is defined by a non-deterministic function that

maps a tuple of the input type of the service to one of possibly many tuples of the output type. There are several reasons why the result might not be functionally dependent on the input. One of them is that the services can have an internal state which influences its result. Also the service can use randomized approximation algorithms, which is often the case in bioinformatics. Finally, the service can be based on a database which is constantly updated. So it seems inappropriate to model services with deterministic functions in the description of Taverna's semantics. Therefore we associate with each label $s \in TS$ a relation $\mathcal{F}[s] \subseteq [\![type_i(s)]\!] \times [\![type_o(s)]\!]$ such that for each tuple $t \in [\![type_i(s)]\!]$ there is at least one tuple $t' \in [\![type_o(s)]\!]$ such that $(t, t') \in \mathcal{F}[s]$. It should be noted at this point that the current implementation of Taverna does not check if a service call returns a tuple with fields of the correct type, but we chose not to model this in the presented formal semantics.

2.4 Scufl syntax

A brief and informal introduction to Scufl has already been given in Section 1.2.1. Here we follow with additional example and formal definitions.

The new example is presented in Fig. 2.1. We start with the analysis of the top Scufl COSW graph which may seem incomplete because the $nin2$ has no incoming data edges. For that port a default value is specified, but that is not visible in the graphical representation.

Another thing that the diagram does not show are the *product strategies* associated with all processors. Such strategies are needed because of the implicit iteration semantics of Scufl that was illustrated by the first processor in Fig. 1.1 (a). In general the implicit iteration strategy states that if a processor receives a value that is nested deeper than expected, it will iterate over subvalues of the expected nesting depth and combine the results again in a list. For example, if a processor that computes a function $f : [\![\langle\, a : \mathcal{M} \,\rangle]\!] \rightarrow [\![\langle\, b : \mathcal{M} \,\rangle]\!]$ receives on its port labeled a the value [“foo”, “bar”], then it will compute the list $[f(\langle\, a = \text{“foo”} \,\rangle), f(\langle\, a = \text{“bar”} \,\rangle)]$. If a processor computes a function that expects many inputs such as $g : [\![\langle\, a : \mathcal{M}, b : \mathcal{M} \,\rangle]\!] \rightarrow [\![\langle\, c : \mathcal{M} \,\rangle]\!]$ and is presented with lists of mime values, then a product strategy such as *cross product* or *dot product* is required to indicate how the input lists are combined into a single list of tuples that represent the combination of complex values to which the function is applied during the iteration. If the list on port a is [“foo”, “bar”] and the list on port b is [“x”, “y”, “z”] then the

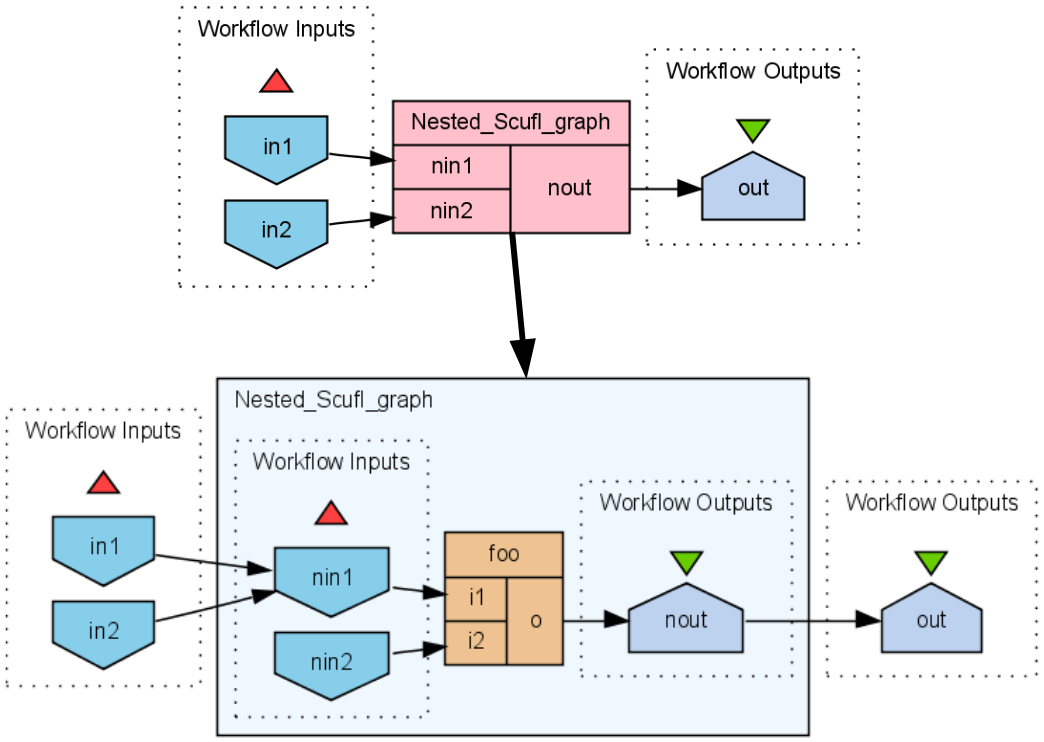

Figure 2.1: An example of a nested Scufl COSW graph

cross product combines them into $[[\langle a = \text{“foo”}, b = \text{“x”}\rangle, \langle a = \text{“foo”}, b = \text{“y”}\rangle, \langle a = \text{“foo”}, b = \text{“z”}\rangle], [\langle a = \text{“bar”}, b = \text{“x”}\rangle, \langle a = \text{“bar”}, b = \text{“y”}\rangle, \langle a = \text{“bar”}, b = \text{“z”}\rangle]]$ and the dot product combines them into $[\langle a = \text{“foo”}, b = \text{“x”}\rangle, \langle a = \text{“bar”}, b = \text{“y”}\rangle]$. For an arbitrary number of input ports a product strategy is defined by an expression in the following syntax:

$$ps \quad ::= \quad \varepsilon \mid PL \mid (ps \otimes ps) \mid (ps \odot ps)$$

in which each label in PL appears at most once. In this expression ε denotes the empty product strategy, a port label product strategy transforms values into tuples, $\otimes$ represents the cross product[1] and $\odot$ represents the dot product. The set of all product strategies is denoted as PS and the set of port labels used in product strategy ps is denoted as $\mathcal{L}(ps)$, i.e., it is defined such that

[1]For lists the cross product $L_1 \otimes L_2$ is not equivalent with $L_2 \otimes L_1$ because the order of the resulting tuples is not the same, but in Taverna there is also a difference in how the result is nested, as will be explained later on.

$\mathcal{L}(\varepsilon) = \emptyset$, $\mathcal{L}(a) = \{a\}$ and $\mathcal{L}(ps_1 \otimes ps_2) = \mathcal{L}(ps_1 \odot ps_2) = \mathcal{L}(ps_1) \cup \mathcal{L}(ps_2)$. The result of a product strategy ps is always a possibly nested list of tuples with fields $\mathcal{L}(ps)$, e.g., if $ps = (a \otimes b) \odot c$, then this results in a possibly nested list of tuples of the form $\langle\, a = x, b = y, c = z \,\rangle$.

A product strategy could be relevant for our example if the processor that represents a nested Scufl COSW graph had a merge strategy specified for its $nin1$ input port, but expected only a single value and not a list. A further explanation of the default value mapping, the incoming-links strategy and the product strategy is provided in Sections 2.5.3 and 2.5.4 respectively.

The final feature presented by the example in Fig. 2.1 is that Scufl COSW graphs are allowed to be recursively nested. The nested Scufl COSW graph is represented by a processor "Nested_Scufl_graph" and its workflow inputs and outputs match the input ports and output ports of the processor. Nesting a part of a Scufl COSW graph into a processor changes its semantics in two ways. The first is that the nested COSW is not executed until all input ports are ready, and the second is that it will apply the implicit iteration strategy during its execution.

The informal discussion until now was illustrated with a notation that is only one of the ways to represent Scufl COSW graphs and more elaborate representations are available in Taverna, although none of them shows all relevant aspects for understanding the complete semantics of the defined COSW. We follow with a comprehensive formal definition of Scufl COSW graphs — Scufl graphs for short. Since these can be recursively nested it will be an inductive definition. For this definition we postulate a countably infinite set $\mathbb{P}$ that contains all possible processor identifiers that we can use in Scufl graphs.

Definition 2.4.1 (Scufl graph)**.** The set of Scufl graphs $\mathcal{G}$ is defined as the smallest set such that every Scufl graph composed of Scufl graphs in $\mathcal{G}$ is also in $\mathcal{G}$, where such a Scufl graph is defined as a tuple $(I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ such that

- $I \subseteq PL$ is a finite set of labels representing the workflow inputs,
- $O \subseteq PL$ is a finite set of labels representing the workflow outputs,
- $P \subseteq \mathbb{P}$ is a finite set of processors disjoint with I and O,
- $\pi_i \subseteq P \times PL$ a finite set representing processor input ports,

- $\pi_o \subseteq P \times PL$ a finite set representing processor output ports,
- $E_d \subseteq (I \times \pi_i) \cup (\pi_o \times \pi_i) \cup (\pi_o \times O)$ is a set of data edges,
- $E_c \subseteq P \times P$ is a set of control edges,
- $\lambda : P \to (TS \cup \mathcal{G})$ is the processor labeling function, that maps processors to either a service label in TS or a nested Scufl graph such that for every processor $p \in P$ it holds that $I(\lambda(p)) = \{l \mid (p, l) \in \pi_i\}$ and $O(\lambda(p)) = \{l \mid (p, l) \in \pi_o\}$,
- $ils : (\pi_i \cup O) \to \{\mathsf{first}, \mathsf{merge}\}$ gives the incoming-links strategy for every input port of a processor and the workflow outputs,
- $ps : P \to PS$ gives the product strategy for every processor $p \in P$ such that $\mathcal{L}(ps(p)) = \{l \mid (p, l) \in \pi_i\}$,
- $dv : \pi_i \to \mathcal{V}_{tav} \cup \{\bot\}$ gives a default value[2] for each input port, where $\bot$ represents the lack of default value and is only allowed if the port has at least one incoming data edge, i.e., if $dv((p, l)) = \bot$, then there is a data edge $(x, (p, l)) \in E_d$ for some x,
- there are no cycles in the *dependency graph* which is defined as a directed graph over P such that there is an edge (p_1, p_2) iff there is a control edge $(p_1, p_2) \in E_c$ or there is a data edge of the form $((p_1, l_1), (p_2, l_2)) \in E_d$,

where the I and O functions for labels in TS are generalized for Scufl graphs such that for a Scufl graph g we let $I(g)$ and $O(g)$ denote the I and O component of g, respectively.

The restriction that a default value must be specified for input ports that have no arriving data edges is more strict than in the real Taverna 1.7.1, where basic processors are allowed to have input ports with neither an incoming data edge nor a data value. This is an often used feature since basic processors can wrap a service with many optional arguments and flags. However, for the sake of simplicity of presentation we will assume that this is represented in the formal syntax by a basic processor that has exactly the

[2]In Taverna 1.7.1, the version that was investigated for this thesis, only strings were allowed as default values.

set of input ports that are provided and has the semantics that the real basic processor has for that particular set of input ports.

Next to generalizing I we also extend the function $type_i$ to Scufl graphs, i.e., $type_i : (TS \cup \mathcal{G}) \to \mathcal{T}_{tup}$. The main purpose of this type is to allow a processor, that is labeled by λ with a Scufl graph, to determine what type it actually expects, and use that to see if for a given complex value it will do an implicit iteration or pass it on to the nested graph. Recall that if a processor receives a value that is nested deeper than expected, then it will identify the subvalues of the expected nesting depth and iterate over those, i.e., pass them on one by one to the nested Scufl graphs.

Informally, the input type of each workflow input is computed by taking the maximum of the types of processor input ports in the nested Scufl graph to which it is connected. So, for example, if the workflow input is connected to two processor input ports that expect $[[\mathcal{M}]]$ and $[\mathcal{M}]$, then the Scufl graph is assumed to expect the type $[[\mathcal{M}]]$ on this input port. The justification for taking the maximum is that this way the processor that contains the nested Scufl graph only starts implicit iteration if it is really necessary, i.e., none of the nested processors to which the value is passed on can deal with it without iteration. For example, assume that the workflow input is connected to a service with input type $\langle\, genes : [\mathcal{M}] \,\rangle$ that expects a list of genes encoded as DNA strands and selects the shortest one. Also assume that another service with input type $\langle\, gen : \mathcal{M} \,\rangle$ is also connected to this workflow input. Then, if the Scufl graph is given a list of genes, the implicit iteration is only needed for the second service and not the whole Scufl graph. This way the first service can find the shortest gene in the whole input list and not in every singleton list resulting from implicit iteration on the workflow input.

Formally, following the induction of $\mathcal{G}$, the input type of a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ with $I = \{l_1, \ldots, l_n\}$, is defined as $type_i(g) = \langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$, where $\tau_i = \mathsf{max}(\{\sigma(l') \mid (l_i, (p, l')) \in E_d, \sigma = type_i(\lambda(p))\})$. Note, that this is well defined since the domain of $type_i(\lambda(p))$ is $I(\lambda(p))$, which by the definition of Scufl graph is equal to $\{l' \mid (p, l') \in \pi_i\}$.

2.4.1 Hierarchically nested Scufl graphs

The Scufl graph definition is an inductive definition that builds larger Scufl graphs by using smaller ones as labels of its processors, i.e., as nested Scufl graphs. It allows us to define notions and prove theorems with induction on the structure of a Scufl graph. Over the set of all Scufl graphs $\mathcal{G}$ we can

define *the nesting graph* that indicates which Scufl graph is nested in which Scufl graph as follows.

Definition 2.4.2 (The nesting graph)**.** *The nesting graph* is the directed edge-labeled graph $\mathcal{N} = (\mathcal{G}, E)$ where $\mathcal{G}$ is the set of nodes and the set of edges $E \subseteq \mathcal{G} \times \mathbb{P} \times \mathcal{G}$ is defined such that $(g, p, g') \in E$ iff $\lambda(p) = g'$ with λ the labeling function of g and p a processor in g.

It is easy to see that, since $\mathcal{G}$ is required in its definition to be minimal, there are no directed cycles in $\mathcal{N}$. The *set of subgraphs of a Scufl graph* g, denoted as $\mathcal{G}_g$, is defined as the set of nodes reachable in $\mathcal{N}$ from g, including g itself. The *nesting graph for a particular Scufl graph* g is denoted as $\mathcal{N}_g$ and defined as subgraph of $\mathcal{N}$ induced by $\mathcal{G}_g$.

It is allowed that the same Scufl graph is reused as a label of more than one processor in a certain Scufl graph definition, either within the same subgraph or in different subgraphs. However, the definition of a state of a Scufl graph can be simplified if such reuse is not allowed and therefore we introduce the notion of *hierarchically nested Scufl graphs*.

Definition 2.4.3 (Hierarchically nested Scufl graphs)**.** A Scufl graph g is said to be *hierarchically nested* iff $\mathcal{N}_g$ is a tree.

Observe that if g is a hierarchically nested Scufl graph then all Scufl graphs in $\mathcal{G}_g$ are also necessarily hierarchically nested.

If a Scufl graph is not hierarchical then it can be made so by replacing each occurrence of a certain Scufl graph with a different but isomorphic Scufl graph. For example, if processors p_1 and p_2 are both labeled with a Scufl graph g, i.e., $\lambda_1(p_1) = \lambda_2(p_2)$, where λ_1 and λ_2 are the processor labeling function of the subgraphs in which p_1 and p_2 appear respectively, then we redefine λ_1 and λ_2 such that $\lambda_1(p_1) = g_1$ and $\lambda_2(p_2) = g_2$, where g_1 and g_2 are different but isomorphic copies of g that do not appear as subgraphs themselves. If we start with a certain Scufl graph and repeat this for every two different processors in subgraphs that are labeled with the same Scufl graph, then we will obtain an equivalent hierarchically nested Scufl graph.

In the remainder of this chapter, where we describe the semantics of Scufl graphs, we will do this only for hierarchically nested Scufl graphs, and therefore, when we refer to a Scufl graph, we always mean a hierarchically nested Scufl graph. The semantics of other Scufl graphs is then defined as the semantics of the corresponding hierarchically nested Scufl graphs. The reason

for this is that in a hierarchically nested Scufl graph we can describe the total state as a mapping of each Scufl graph that it contains to its particular state. The exponential blow-up that can be caused by making a Scufl graph hierarchical, is in some sense unavoidable, because it is linked to the potentially exponential number of Scufl graph instances for which a state has to be described.

2.5 Processor execution

2.5.1 An overview of processor execution

A successful execution of a processor is a complex event best explained by dividing it in several steps. We give here an informal overview of those steps and discuss the first two of them in the rest of this section in further detail by defining the functions that compute them. Then, in Section 2.6, using those functions and additional prerequisites defined in Section 2.5.2, we discuss the execution of a Scufl graph as a whole, look into all the steps together, and explore all possible scenarios including the possibility of processor failure.

We now proceed with the informal description of the steps of a successful execution of a processor:

Computing the values in the input ports

In the first step an input value for each input port is computed from the values that were sent to it through the incoming data edges. This is done by combining these values into a single complex value according to the incoming-links strategy. The select-first strategy simply takes the first value that arrives and ignores the others, and the merge strategy creates a list containing all the arrived values.

Combining the input port values into the processor input value

A *processor input value* is computed, which is a single tuple that can be processed by the service that the processor represents, or a possibly nested nested list of such tuples. If for every input port of the processor the value computed in the previous step is of the type expected by the processor, i.e., is not overly nested, then the processor input value is a tuple labeled by input port labels and holding the input port values. For example, if the input

ports are labeled a and b and their computed input port values v_a and v_b are of the expected type, then the processor input value is $\langle\, a : v_a, b : v_b \,\rangle$. If any of the values computed in the preceding steps is too deeply nested, then the values of the different input ports must be combined into a single nested value, i.e., a list of tuples over which the processor can iterate. For example, assume that v_a is a list of mime values and v_b is a list of lists, while the processor expects types $\mathcal{M}$ and $[\mathcal{M}]$, respectively. The computation of the processor input value can then be thought of as consisting of two steps. First, the values that were computed for the input ports are transformed into values where the subvalues of the type that is expected are identified by packing them in singleton tuples. Continuing the last example, the value for the input port labeled a would be transformed to a list of tuples of type $\langle\, a : \mathcal{M} \,\rangle$ and the value for the input port labeled b would be transformed to a list of tuples of type $\langle\, b : [\mathcal{M}] \,\rangle$. Second, the product strategy of the processor describes which combinations of the identified tuples are taken and how they are nested in the result. For example, a strategy consisting of a single cross product will combine all tuples in the first value with all tuples in the second, resulting in a doubly nested list of tuples of type $\langle\, a : \mathcal{M}, b : [\mathcal{M}] \,\rangle$.

Performing the execution or the iteration

If the value computed in the preceding step is a tuple, the processor is executed once, producing one result tuple with values for every output port. If the processor input value is a list, it is iterated over by executing the processor for each tuple in it. The result for each output port contains a list of values from result tuples of subsequent iteration steps that is structured accordingly to the nesting structure of the processor input list. Following the previous example, if the processor has two output ports labeled c and d, and is associated with a Taverna service with output type $\langle\, c : [\mathcal{M}], d : \mathcal{M} \,\rangle$, then the iteration will produce a list of lists with elements of type $[\mathcal{M}]$ for port labeled c, and a list of lists with elements of type $\mathcal{M}$ for port labeled d.

Copying the computed output port values

When the normal execution or iteration has finished the values computed in the processor output ports are copied to all processor input ports and workflow outputs to which they are connected.

2.5.2 Extended complex value construction and deconstruction

As explained in the informal description of the semantics of processor execution in Section 2.5.1, we can describe the execution of a processor after the processor input value has been computed as a process that takes a possibly nested list of tuples, iterates over all tuples by executing the processor and while doing so constructs for each output port a value by inserting, at the position of the original tuple, the value that was computed for that output port by the iteration step.

Since $\mathcal{V}_{tup}$ includes tuples, but not lists of tuples, we define an extended complex value set $\mathcal{V}_{ext}$ as the smallest set such that (1) $\mathcal{V}_{tup} \subseteq \mathcal{V}_{ext}$ and (2) if $x_1, \ldots, x_n \in \mathcal{V}_{ext}$ then the list $[x_1, \ldots, x_n]$ is in $\mathcal{V}_{ext}$.

In order to identify the position of tuples and other subvalues in an extended complex value we introduce the notion of subvalue index. By a *subvalue* of an extended complex value v we mean v itself, any element of v, any element of element of v, and so on, up to the tuples. For example, if $v = [[a, b], [c]]$, where a, b and c are tuples, then all subvalues of v are: v, $[a, b]$, $[c]$, a, b and c.

Definition 2.5.1 (Subvalue index)**.** A *subvalue index*, or simply *index*, is a list of positive natural numbers. Such indices are denoted by a list of numbers separated by slashes, e.g., 2/3/8 and 1/1, and the empty list is denoted as ϵ. The set of all complex value indices is denoted as $\mathcal{I}$.

The numbers in an index are listed from most significant on the left, to the least significant on the right. Following the last example, the subsequent indexes of the mentioned subvalues of v are: ϵ, 1, 2, 1/1, 1/2 and 2/1.

Formally, the subvalue indicated by an index is defined by the function $\mathsf{get} : \mathcal{V}_{ext} \times \mathcal{I} \to (\mathcal{V}_{ext} \cup \bot)$ such that $\mathsf{get}(v, \epsilon) = v$, and $\mathsf{get}(v, i/\alpha) = \mathsf{get}(v_i, \alpha)$ if $v = [v_1, \ldots, v_n]$ and $1 \leq i \leq n$, and $\mathsf{get}(v, i/\alpha) = \bot$ otherwise. For example, if $v = [[a, b], [c]]$, then $\mathsf{get}(v, 2/1) = c$ and $\mathsf{get}(v, 2/2) = \bot$.

We assume that complex value indices are ordered according to the lexicographical ordering, i.e., the smallest binary relation $\preceq$ over $\mathcal{I}$ such that for every $i, j \in \mathbb{N}$ and $\alpha, \beta \in \mathcal{I}$ it holds that (1) $\epsilon \preceq \alpha$, (2) if $i \leq j$, then $i/\alpha \preceq j/\beta$ and (3) if $\alpha \preceq \beta$, then $i/\alpha \preceq i/\beta$. As usual this defines a linear order over $\mathcal{I}$.

In order to be able to iterate over all tuples in an extended complex value we define a function that retrieves the index of the first tuple and a function to

jump to the index of the next tuple. The first function is $\mathsf{first} : \mathcal{V}_{ext} \rightarrow (\mathcal{I} \cup \bot)$ which is defined such that $\mathsf{first}(v) = \alpha$ where α is the smallest index such that $\mathsf{get}(v, \alpha) \in \mathcal{V}_{tup}$, and $\mathsf{first}(v) = \bot$ if there is no such α. The second function is $\mathsf{next} : \mathcal{V}_{ext} \times \mathcal{I} \rightarrow (\mathcal{I} \cup \bot)$ and is defined such that $\mathsf{next}(v, \alpha) = \beta$ if β is the smallest index larger than α such that $\mathsf{get}(v, \beta) \in \mathcal{V}_{tup}$, and $\mathsf{next}(v, \alpha) = \bot$ if such a β does not exist.

Finally, we define a function $\mathsf{put}(v, \alpha, w)$ that inserts into the complex value v at position α the complex value w, which can be used to construct complex values. For example, $\mathsf{put}([x, [y]], 2/1, z) = [x, [z]]$ and $\mathsf{put}([], \epsilon, z) = z$. If the position α does not yet exist in v then it is extended minimally with empty lists to create it. For example, $\mathsf{put}([], 1/1/1, x) = [[[x]]]$ and $\mathsf{put}([], 2/1, x) = [[], [x]]$. Formally, this function $\mathsf{put} : \mathcal{V}_{tav} \times \mathcal{I} \times \mathcal{V}_{tav} \rightarrow \mathcal{V}_{tav}$ is defined such that (1) $\mathsf{put}(v, \epsilon, w) = w$, (2) $\mathsf{put}(v, i/\alpha, w) = \mathsf{put}([], i/\alpha, w)$ if $v \in \mathcal{V}_{\mathcal{M}}$, (3) $\mathsf{put}([], 1/\alpha, w) = [\mathsf{put}([], \alpha, w)]$, (4) $\mathsf{put}([v] + v', 1/\alpha, w) = [\mathsf{put}(v, \alpha, w)] + v'$, (5) $\mathsf{put}([], i/\alpha, w) = [[]] + \mathsf{put}([], (i-1)/\alpha, w)$ if $i > 1$, (6) $\mathsf{put}([v] + v', i/\alpha, w) = [v] + \mathsf{put}(v', (i-1)/\alpha, w)$ if $i > 1$.

2.5.3 Incoming-links strategy semantics

Here we define the semantics of incoming-links strategy expressions which are used to indicate how to compute the value for a processor input port or workflow output by composing it from values provided from multiple incoming data edges. The computation is done incrementally, that is, a temporary result is extended each time a new value arrives from one of the data edges that did not already supply a value. The lack of a previous temporary value at the start of the process is represented by $\bot$.

The select-first incoming-links strategy picks the first value to arrive and ignores all the other. This is the default behavior of processor input ports and workflow outputs. The function $[\![\mathsf{first}]\!] : ((\mathcal{V}_{tav} \cup \{\bot\}) \times \mathcal{V}_{tav}) \rightarrow \mathcal{V}_{tav}$ takes as the first argument the current temporary result and as the second the value provided by the next data edge. As a result the new temporary result is returned. Formally:

$$[\![\mathsf{first}]\!](t, v) = \begin{cases} v & \text{if } t = \bot \\ t & \text{otherwise} \end{cases}$$

The merge incoming-links strategy combines all incoming values as elements of a list. It was added to Taverna 1.3.1 to prevent the need for creation

of user defined n-argument processors that compose their arguments into a list. As with select-first, the merge function $[\![\mathsf{merge}]\!] : ((\mathcal{V}_{tav} \setminus [\![\mathcal{M}]\!] \cup \{\bot\}) \times \mathcal{V}_{tav}) \to \mathcal{V}_{tav}$ has two arguments, yet now the temporary value is never of type $\mathcal{M}$ since it is a list of values provided so far. Formally:

$$[\![\mathsf{merge}]\!](t, v) = \begin{cases} [v] & \text{if } t = \bot \\ t + [v] & \text{otherwise} \end{cases}$$

Strictly speaking this is not a merge, but we adhere to the Taverna terminology.

2.5.4 Product strategy semantics

Here we define the semantics of product strategy expressions $ps \in PS$. The product strategy expressions are used to transform values from $\mathcal{V}_{tav}$, that are provided on individual input ports of a given processor p, to extended complex values that contain tuples of type $type_i(\lambda(p))$, i.e., lists of tuples ready to be iterated upon by p.

The values provided on a processors' input ports have to be combined into a processor input value that is either a single tuple which can be processed by the service that the processor represents or a nested list of such tuples. This is done in two steps. The first step transforms each of the values provided on every input port into a single unary tuple or a list of unary tuples. The tuples' field is labeled with the same label as the respective input port and they contain values of the type that is expected on that port. The second step combines such preprocessed values for processors with multiple input ports into a single n-ary tuple or a nested list of those.

We now describe the first step in more detail. Its purpose is to identify the subvalues that are of a nesting depth acceptable by the processor. For example, if the value on the input port with label a is $[[1, 2], [], [3]]$ and the processor expects a value of type $[\mathcal{M}]$ on it, then the value is transformed to $[\langle\, a = [1, 2]\,\rangle, \langle\, a = []\,\rangle, \langle\, a = [3]\,\rangle]$. If this is the only input port, then the processor will iterate over the three values $[1, 2]$, $[]$ and $[3]$. If, on the other hand, value of type $\mathcal{M}$ is expected, then it is transformed to $[[\langle\, a = 1\,\rangle, \langle\, a = 2\,\rangle], [], [\langle\, a = 3\,\rangle]]$ and the processor will iterate over the three values 1, 2 and 3. This is formalized by the packing function $\mathsf{pack}_{l:\tau} : \mathcal{V}_{tav} \to \mathcal{V}_{ext}$ that identifies nested values of type τ and packs them into tuples of type $\langle\, l : \tau\,\rangle$.

Formally, it is defined as follows:

$$\mathsf{pack}_{l:\tau}(x) = \begin{cases} \langle\, l = x \,\rangle & \text{if } x \in [\![\tau]\!] \\ [\mathsf{pack}_{l:\tau}(x_1), \ldots, \mathsf{pack}_{l:\tau}(x_n)] & \text{if } x = [x_1, \ldots, x_n] \notin [\![\tau]\!] \end{cases}$$

This function is well defined for every $x \in \mathcal{V}_{tav}$, which can be shown with induction on the structure of x and using the fact that $\mathcal{V}_{\mathcal{M}} \subseteq [\![\tau]\!]$ for any $\tau \in \mathcal{T}_{tav}$. It is possible that a value of type τ contains a nested value that is also of type τ. For example, if $\tau = [[\mathcal{M}]]$ and $x = [[[1]]]$, then there are in x three nested values of type τ, namely 1, [1] and [[1]]. In that case the nested value with the largest nesting depth is chosen and so $\mathsf{pack}_{a:\tau}(x) = [\langle\, a = [[1]] \,\rangle]$. For a more elaborate example consider:

$$\begin{aligned} &\mathsf{pack}_{a:[\mathcal{M}]}([[1],[[2],3],4]) \\ &\quad = [\mathsf{pack}_{a:[\mathcal{M}]}([1]), \mathsf{pack}_{a:[\mathcal{M}]}([[2],3]), \mathsf{pack}_{a:[\mathcal{M}]}(4)] \\ &\quad = [\langle\, a = [1] \,\rangle, [\mathsf{pack}_{a:[\mathcal{M}]}([2]), \mathsf{pack}_{a:[\mathcal{M}]}(3)], \langle\, a = 4 \,\rangle] \\ &\quad = [\langle\, a = [1] \,\rangle, [\langle\, a = [2] \,\rangle, \langle\, a = 3 \,\rangle], \langle\, a = 4 \,\rangle] \end{aligned}$$

Note, that the values 3 and 4 are in $[\![[\mathcal{M}]]\!]$ and therefore also packed in a tuple.

We now proceed to the second step where we deal with the case of processors with multiple input ports. There the extended complex values computed by the packing function have to be combined. For this the cross and dot product strategy expressions are used to represent the $\times$ — cross and $\cdot$ — dot product functions[3]. An intuition of how they work on flat lists has already been given in Section 2.4.

For higher level lists the dot product used in Taverna fully flattens its arguments, operates on the flat lists and structures the result according to the structure of the argument with the highest nesting depth. For example, if a, b, c, d and e are tuples, then $[a, b] \cdot [[c], [d, e]] = [[a \cup c], [b \cup d]]$, where the union of tuple values is a well defined tuple since in product strategy expressions each label from PL appears at most once. In the case where both arguments have the same nesting depth the structuring occurs with respect to the left one. For the formal definition of the dot product we define three auxiliary notions.

[3]The functions $\times$ and $\cdot$ should not be confused with $\otimes$ and $\odot$, which are the corresponding syntactical constructs in product strategy expressions.

The first is the function flat^* that flattens values in $\mathcal{V}_{ext}$, i.e., recursively nested lists of tuples, to lists of tuples, e.g, if x_1, x_2 and x_3 are tuples, then $\mathsf{flat}^*([[[x_1]], [[x_2], [x_3]]]) = [x_1, x_2, x_3]$. Formally, it is defined such that:

$$\mathsf{flat}^*(x) \quad = \quad \begin{cases} [] & \text{if } x = [] \\ [x] & \text{if } x \in \mathcal{V}_{tup} \\ \mathsf{flat}^*(x_1) + \ldots + \mathsf{flat}^*(x_n) & \text{if } x = [x_1, \ldots, x_n] \end{cases}$$

The second notion is that of *the tuple nesting depth* of a value x in $\mathcal{V}_{ext}$, denoted as $\mathsf{tnd}(x)$, which can be informally described as the maximum nesting depth of tuples in x. It is formally defined such that (1) $\mathsf{tnd}(x) = 0$ for $x \in \mathcal{V}_{tup}$, (2) $\mathsf{tnd}([]) = 1$, and (3) $\mathsf{tnd}([x_1, \ldots, x_n]) = 1 + max_{1 \leq i \leq n}(\mathsf{tnd}(x_i))$.

Finally, a $\mathsf{replace} : \mathcal{V}_{ext} \times \mathcal{V}_{ext} \rightarrow \mathcal{V}_{ext}$ partial function is defined which replaces all the subsequent tuple subvalues in the complex value provided as the first argument with the subsequent elements of the tuple list provided as the second argument. For example, assuming that every z_i and t_i is a tuple, $\mathsf{replace}([[z_1, z_2], [z_3]], [t_1, t_2, t_3]) = [[t_1, t_2], [t_3]]$. Additionally, if the first argument has more tuples than the second, the extra ones are ignored, e.g., $\mathsf{replace}([[z_1], [z_2, z_3], [z_4]], [t_1, t_2]) = [[t_1], [t_2]]$. Similarly, we also ignore its subvalues containing no tuples at all, e.g., $\mathsf{replace}([[z_1, z_2], [z_3], []], [t_1, t_2, t_3]) = [[t_1, t_2], [t_3]]$, but only if it does not change the positions of the other subvalues, e.g., $\mathsf{replace}([[[z_1], [z_2]], [[]], [[z_3, z_4]]], [t_1, t_2, t_3]) = [[[t_1], [t_2]], [], [[t_3]]]$. Formally, if z is a complex value such that $\mathsf{flat}^*(z) = [z_1, \ldots, z_m]$ and $t = [t_1, \ldots, t_n]$ where $m \geq n$, then $\mathsf{replace}(z, t) = r$ where r is the smallest complex value such that $\mathsf{flat}^*(r) = [r_1, \ldots, r_n]$ and $\mathsf{get}(r, \alpha_i) = r_i$ for all $i = 1 \ldots n$ and $\alpha_1, \ldots, \alpha_n$ being the respective indexes of $z_1, \ldots, z_n$ in z. The ordering of the complex values that we refer to in this definition is given such that: (1) if a and b are tuples, then $a \leq b$ iff $a = b$, and (2) $[a_1, \ldots, a_n] \leq [b_1, \ldots, b_m]$ iff $n \leq m$ and for each $i = 1, \ldots, n$ it is true that $a_i \leq b_i$. It is easy to see, that this indeed defines a partial order.

With these notions we can now define the dot product. Let x and y be complex values such that $\mathsf{flat}^*(x) = [x_1, \ldots, x_n]$ and $\mathsf{flat}^*(y) = [y_1, \ldots, y_m]$. The dot product function $\cdot : \mathcal{V}_{ext} \times \mathcal{V}_{ext} \rightarrow \mathcal{V}_{ext}$ is defined such that $x \cdot y = \mathsf{replace}(z_{x,y}, t_{x,y})$ where $t_{x,y} = [x_1 \cup y_1, \ldots, x_{min(n,m)} \cup y_{min(n,m)}]$ and $z_{x,y} = y$ if $\mathsf{tnd}(x) < \mathsf{tnd}(y)$ and $z_{x,y} = x$ otherwise. It is easy to see that $t_{x,y}$ and $z_{x,y}$ are well defined, and because $n \geq min(n, m) \leq m$ so is the dot product.

Note that the pruning of the nested lists with no tuples by the $\mathsf{replace}$ function is consistent with how Taverna works, e.g., for tuples a, b, c, d and e,

it holds in Taverna that $[[[]],[[a,b]]] \cdot [c,d,e] = [[],[[a \cup c],[b \cup d]]]$. Also note that because of how $z_{x,y}$ is defined in the dot product function definition it is the tuple nesting depth of the arguments that decides which of the two arguments will determine the nesting structure of the result, as indeed is the case in Taverna. An interesting alternative might be to always let the left argument determine the nesting structure. That way the user can control this by simply changing the order in the product strategy expression.

The generalization of the dot product in Taverna is not the only possible generalization and may sometimes lead to unexpected results. To illustrate this we propose here an alternative where the dot product is generalized recursively. For example, if $x = [x_1, x_2]$ and $y = [y_1, y_2, y_3]$, then $x \cdot_r y = [x_1 \cdot_r y_1, x_2 \cdot_r y_2]$. If $x = [x_1, x_2]$ and y is a tuple, then $x \cdot_r y = [x_1 \cdot_r y]$, and if both x and y are tuples, then $x \cdot_r y = x \cup y$. Formally, we define the recursive dot product function $\cdot_r : \mathcal{V}_{ext} \times \mathcal{V}_{ext} \rightarrow \mathcal{V}_{ext}$ as follows:

$$k \cdot_r l \quad = \quad \begin{cases} [] & \text{if } \mathsf{flat}^*(k) = [] \text{ or } \mathsf{flat}^*(l) = [] \\ [k_1 \cdot_r l] & \text{if } k = [k_1, \ldots, k_n] \text{ and } l \in \mathcal{V}_{tup} \\ [k \cdot_r l_1] & \text{if } k \in \mathcal{V}_{tup} \text{ and } l = [l_1, \ldots, l_m] \\ [k_1 \cdot_r l_1, \ldots, k_{min(n,m)} \cdot_r l_{min(n,m)}] & \text{if } k = [k_1, \ldots, k_n] \text{ and } l = [l_1, \ldots, l_m] \\ k \cup l & \text{if } k \in \mathcal{V}_{tup} \text{ and } l \in \mathcal{V}_{tup} \end{cases}$$

To motivate the alternative definition let us analyze a simple example from

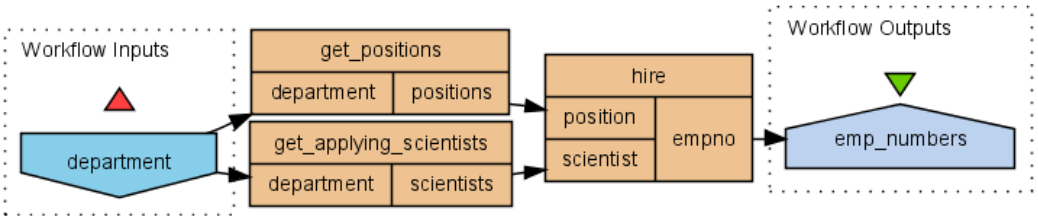

Figure 2.2: Recursive dot product motivation example

Fig. 2.2 where the initial value with an university department identifier, e.g., "informatics", is used by two services, of which one produces a list of positions available in this department and other a list of scientists applying for work there. The list of positions is sorted by their appeal and the scientists are sorted according to their achievements. A third service is used to hire a scientist for a position. To deal with the values of higher types it uses the

dot product strategy. This way the best positions are assigned to the best scientists and the hiring occurs while both positions and scientists are still available. Observe now that if this Scufl graph is executed with a list of departments identifiers, e.g., ["physics", "bioinformatics", "informatics"] and the implicit iteration over "get_positions" and "get_applying_scientists" returned $p = [[pp_1, pp_2], [pb_1, pb_2, pb_3], [pi_1, pi_2]]$ and $s = [[sp_1, sp_2, sp_3], [sb_1], [si_1, si_2]]$ respectively, then the dot product of Taverna intermixes position and scientists from different departments, i.e., the worst physicist sp_3 will be hired on the best bioinformatics position pb_1 and the best informatician si_1 will be hired on the worst bioinformatics position pb_3. Even if it is the case that informaticians and especially physicists do well as bioinformaticians, the informatics department becomes undermanned and does not get the best people. Clearly our recursive definition of dot product does not intermix the values, so scientists will only be hired by the departments they applied to and the departments will be able to hire all the scientists that applied to them as long as they have enough positions.

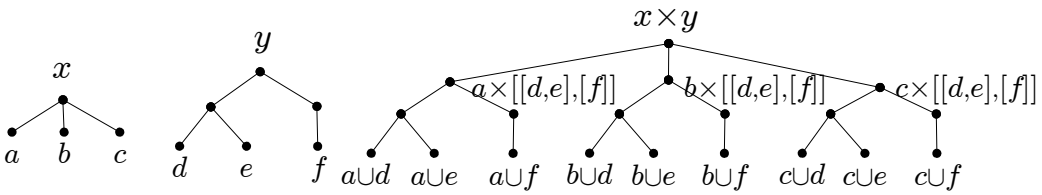

Figure 2.3: Cross product for higher list types

To understand the cross product of Taverna for higher list types it is convenient to think of the nested lists as ordered trees with the leaves labeled with tuple values. A tree interpretation of values $x = [a, b, c]$ and $y = [[d, e], [f]]$, where a, b, c, d, e and f are tuples, is given in Fig. 2.3. The cross product of x and y is then obtained by replacing each of the leaf tuples t_x in x by a copy of the y tree that in turn has its every leaf tuple value t_y replaced by $t_x \cup t_y$ (see Fig. 2.3). This in our case results in $[[[a \cup d, a \cup e], [a \cup f]], [[b \cup d, b \cup e], [b \cup f]], [[c \cup d, c \cup e], [c \cup f]]]$. Formally, we define the cross product function $\mathcal{V}_{ext} \times \mathcal{V}_{ext} \to \mathcal{V}_{ext}$ as follows:

$$k \times l \quad = \quad \begin{cases} [] & \text{if } \mathsf{flat}^*(k) = [] \text{ or } \mathsf{flat}^*(l) = [] \\ [k_1 \times l, \ldots, k_n \times l] & \text{if } k = [k_1, \ldots, k_n] \text{ and } l \neq [] \\ [k \times l_1, \ldots, k \times l_m] & \text{if } k \in \mathcal{V}_{tup} \text{ and } l = [l_1, \ldots, l_m] \\ k \cup l & \text{if } k \in \mathcal{V}_{tup} \text{ and } l \in \mathcal{V}_{tup} \end{cases}$$

Observe that the cross product of Taverna for flat lists is not a natural version of the Cartesian product for lists. Although all the combinations of the argument's tuples are returned, the nesting structure of the result is deeper, i.e., if $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_m]$, then $x \times y = [[x_1 \cup y_1, \ldots, x_1 \cup y_m], \ldots, [x_n \cup y_1, \ldots, x_n \cup y_m]]$, while for the Cartesian product one would expect $[x_1 \cup y_1, \ldots, x_1 \cup y_m, \ldots, x_n \cup y_1, \ldots, x_n \cup y_m]$. A natural generalization of the usual Cartesian product for lists can be obtained by defining it recursively for higher order lists as follows:

$$k \times_r l \quad = \quad \begin{cases} [] & \text{if } \mathsf{flat}^*(k) = [] \text{ or } \mathsf{flat}^*(l) = [] \\ [k_1 \times_r l, \ldots, k_n \times_r l] & \text{if } k = [k_1, \ldots, k_n] \text{ and } l \in \mathcal{V}_{tup} \\ [k \times_r l_1, \ldots, k \times_r l_m] & \text{if } k \in \mathcal{V}_{tup} \text{ and } l = [l_1, \ldots, l_m] \\ [k_1 \times_r l_1, \ldots, k_1 \times_r l_m, & \\ \ \ldots, & \text{if } k = [k_1, \ldots, k_n] \text{ and } l = [l_1, \ldots, l_m] \\ \ k_n \times_r l_1, \ldots, k_n \times_r l_m] & \\ k \cup l & \text{if } k \in \mathcal{V}_{tup} \text{ and } l \in \mathcal{V}_{tup} \end{cases}$$

Notice, that when empty lists don't appear, the nesting depth of the result value for the cross product is the sum of the nesting depths of the arguments and for the generalized Cartesian product it is the maximum. We want to stress that the summing of nesting depths of the arguments in the cross product used in Taverna may be sometimes unexpected for the user. For example, when a Scufl graph with one input port of type $\mathcal{M}$ and one output port type $\mathcal{M}$ is initiated with a list of lists of mime elements, then most users would expect for it to result also with such a list. Yet, if at the start of this Scufl graph a preprocessing of the input value takes place by a binary operation for which a cross product is specified and both input ports are connected to the workflow input, then the result will be a four times nested list of mime elements. Even more interesting is the observation that this will not be the case when such a Scufl graph is nested. Then, a full implicit iteration will occur for the processor representing the nested Scufl graph, i.e., the nested Scufl graph is executed on values of the expected type and the implicit iteration mechanism collects the results into a list of the same structure as the one that was iterated over.

Besides the different nesting of result values, the cross product of Taverna and the generalized Cartesian product order the leaf elements differently, e.g., if a, b, c and d are tuples, $x = [[a, b]]$, and $y = [[c], [d]]$, then $\mathsf{flat}^*(x \times y) = [a \cup c, a \cup d, b \cup c, b \cup d]$, while $\mathsf{flat}^*(x \times_r y) = [a \cup c, b \cup c, a \cup d, b \cup d]$.

Both operations, the cross product and the recursively generalized Cartesian product, may be useful to the user and it is not obvious how to simulate one with the other.

Given the definitions of the cross and dot product we can now define the semantics and typing of a product strategy ps for a processor in a certain Scufl graph. Let τ be the input tuple type of the processor and ps a product strategy such that $\mathcal{L}(ps) = \mathrm{dom}(\tau)$. Then we define for each such product strategy ps and type τ a function $[\![ps]\!]^\tau : (\mathcal{L}(ps) \to \mathcal{V}_{tav}) \to \mathcal{V}_{ext}$ that maps a tuple of complex values containing a field for each port label in ps to an extended complex value over which the processor can execute or iterate. Formally, we define this function as follows:

$$\begin{aligned}
[\![\varepsilon]\!]^\tau(t) &= \langle\rangle \\
[\![l]\!]^\tau(t) &= \mathsf{pack}_{l:\tau(l)}(t(l)) \\
[\![(ps_1 \otimes ps_2)]\!]^\tau(t) &= [\![ps_1]\!]^\tau(t|_{\mathcal{L}(ps_1)}) \times [\![ps_2]\!]^\tau(t|_{\mathcal{L}(ps_2)}) \\
[\![(ps_1 \odot ps_2)]\!]^\tau(t) &= [\![ps_1]\!]^\tau(t|_{\mathcal{L}(ps_1)}) \cdot [\![ps_2]\!]^\tau(t|_{\mathcal{L}(ps_2)}).
\end{aligned}$$

All versions of cross and dot products defined here are binary expressions. They can be easily generalized for more arguments thanks to the observation that $x \times (y \times z) = (x \times y) \times z$ and $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ regardless of which, original or recursive, definition is chosen. In fact the generalized versions are available in Taverna. Note also, that for higher level lists usually $x \times y \neq y \times x$, so the order of port labels in the product strategy expression is important.

2.6 Transition system semantics

In this section we define the semantics of Scufl graphs in terms of a transition system, i.e., we specify a set of possible states of the Scufl graph and which transitions are possible between these states. The following subsection discusses the states, it is followed by subsections on auxiliary notions for describing the transitions, then the transitions themselves are discussed, and the final subsection shows that the defined semantics can be used in proofs of properties of Scufl graph.

2.6.1 Scufl graph state

The state of a Scufl graph is described in two levels. At the lowest level we describe the so-called *local state* of each of the subgraphs. This local state

consists of a descriptions of the states of the workflow inputs and outputs, the processor input and output ports, and the processors themselves, but only those that are directly part of the subgraph in question. At the highest level the *global state* of a Scufl graph g is described by simply giving the local states of all the Scufl graphs in $\mathcal{G}_g$, i.e., all subgraphs, including the Scufl graph itself. In the following we first define the notion of local state, followed by a definition of the global state.

We start with an informal introduction of the components of a local state. Consider the Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$. The *workflow input value mapping* $Iv : I \rightarrow (\mathcal{V}_{tav} \cup \perp)$ stores the value associated with each workflow input. The $\perp$ represents the lack of value, which here means that it has not been inserted yet or has already been pushed to the connected processor input ports. Next, the *workflow output value mapping* $Ov : O \rightarrow (\mathcal{V}_{tav} \cup \perp)$, the *input port value mapping* $ipv : \pi_i \rightarrow (\mathcal{V}_{tav} \cup \perp)$ and the *output port value mapping* $opv : \pi_o \rightarrow (\mathcal{V}_{tav} \cup \perp)$ store the values associated with workflow outputs, processor input ports and processor output ports respectively. The stored values are constructed by the incoming-links strategy function (see Section 2.5.3) in case of the workflow output value mapping and the input port value mapping, or by the `put` function (see Section 2.5.2) in case of the output port value mapping. This means that even if they have already been defined, i.e., are not equal to $\perp$, they may still be extended with additional values arriving from further data edges or iteration steps, respectively. Next, each processor itself can be in several states like "scheduled" or "preparing", which is specified by the *execution state mapping* $es : P \rightarrow \{\text{"scheduled"}, \text{"preparing"}, \text{"waiting"}, \text{"finished"}, \text{"failed"}\}$. The state "scheduled" indicates that the processor has not yet been used. The state "preparing" indicates that execution of this processor has already started but the input value, or in case of iteration some of its subvalues, have still to be processed. The state "waiting" indicates that the processor is waiting for a nested Scufl graph or an external service to return a result[4]. The state "finished" indicates that it has finished with success. Finally, "failed" indicates that it has finished with failure. Finally, since a processor might have to iterate, the current position in the iteration is stored by the *iteration index mapping* $ii : P \rightarrow \mathcal{I}$.

[4]In official Taverna terminology the states that we call "preparing" and "waiting" are divided into *executing* and *iterating* for when the processor is either processing a value of its expected type or a value that is more deeply nested, respectively.

Definition 2.6.1 (Local state)**.** Given a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$, a local state of g is a tuple $ls = (Iv, Ov, ipv, opv, es, ii)$ such that:

- $Iv : I \to (\mathcal{V}_{tav} \cup \bot)$ is the workflow input value mapping,
- $Ov : O \to (\mathcal{V}_{tav} \cup \bot)$ is the workflow output value mapping,
- $ipv : \pi_i \to (\mathcal{V}_{tav} \cup \bot)$ is the input port value mapping,
- $opv : \pi_i \to (\mathcal{V}_{tav} \cup \bot)$ is the output port value mapping,
- $es : P \to \{$"scheduled", "preparing", "waiting", "finished", "failed"$\}$ is the processor state mapping,
- $ii : P \to \mathcal{I}$ is the iteration index mapping.

We refer to the set of all local states for all Scufl graphs as LS. The input port value of an input port (p, l), normally denoted as $ipv((p, l))$, will also be written as $ipv(p, l)$. Likewise the output port value of an output port (p, l) will also be written as $opv(p, l)$.

Scufl graphs do not have stateful features, such as counters or data-stores, that can be read and updated during a run of the Scufl graph. So the definition of a local state does not contain anything that represents the state of such elements. Of course these can be simulated by defining a set of special basic processors that have as their semantics that they read or write certain data stores. However, also for such basic processors that represent calls to stateful services, we do not represent the state of the service in the local state. This is because we consider this state not a part of the Taverna system but a part of the environment with which it communicates. It is possible to reason about the behavior of Taverna while taking into account that a service it calls has certain stateful behavior, e.g., is a counter. For that a description of that behavior, ideally also in the form of a state transition system, has to be composed with Taverna's state transition system such that their mutual transitions, i.e., the service calls, are synchronized.

Definition 2.6.2 (Global state)**.** A global state of a Scufl graph g is a function $gs : \mathcal{G}_g \to LS$ that associates with each subgraph $g' \in \mathcal{G}_g$ a local state of g'.

Note that only one state is associated with each subgraph which means that it executes only one run at any moment. Since we restrict ourselves to hierarchically nested Scufl graphs (see Section 2.4.1) this cannot lead to resource contention between different parts of the Scufl graph. Although in Taverna it is possible to choose whether the iteration steps are executed sequentially or in parallel, we will only describe here sequential execution. It is possible to describe a semantics that would allow parallelism, see for example [28, 26], but we have chosen not to do so in this thesis because it would complicate the presentation of the main concepts of the semantics of Scufl.

2.6.2 Ready ports and enabled processors

The fundamental notion that determines the execution of a Scufl graph is the notion of *enabledness* of a processor, i.e., whether in a certain state a processor can start processing its input. One necessary condition for this is that all its input ports are *ready*, i.e., store a fully constructed input value. In the following we describe these two notions in more detail.

Informally, a processor input port is said to be ready, if the value assigned to it will not be further extended by the incoming-links strategy function (see Section 2.5.3).

Definition 2.6.3 (Ready input port)**.** Given a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ we say that input port $p_{in} \in \pi_i$ is *ready* in a local state $ls = (Iv, Ov, ipv, opv, es, ii)$ iff p_{in} either has no incoming data edges or if p_{in} has incoming data edges then it holds that :

(i) if $ils(p_{in}) = \mathsf{first}$, then the first value for p_{in} has already arrived, i.e., $ipv(p_{in}) \neq \perp$, and

(ii) if $ils(p_{in}) = \mathsf{merge}$, then all the values for p_{in} have already arrived, i.e., the $ipv(p_{in})$ is a list with length equal to the number of data edges ending in p_{in}.

Recall that input ports with no incoming edge must have a default value specified, and therefore are always ready.

Note that if a select-first incoming-links strategy is specified, the port does not wait for values from all incoming data edges, but is ready after receiving the first one. On the other hand, if the merge incoming-links strategy is

specified, the port has to wait for a value from every incoming data edge. This way the merge setting can be viewed as a shortcut for an intermediary processor with a separate input port for each incoming data edge and one output port, that composes values from distinct ports into a list[5].

The notion of readiness is extended to workflow outputs, which is natural since the values stored there will also be constructed by the incoming-links strategy function (see Section 2.5.3). There is a small exception to this in the behavior of Taverna 1.7.1, where a workflow output with the merge strategy may become ready even if only values from some of the incoming data edges arrived and it is certain that no more will since the processors that should produce them failed. However, this behavior seems to be idiosyncratic.

The notion of readiness now allows us to define the notion of enabledness. Informally, a processor is said to be enabled, when it can start processing its input. There are three conditions that have to hold for that to happen. First, it has to be scheduled, which means that in the current run of the Scufl graph it was not used yet. Second, all the processors that it synchronizes with through the control edges must have already finished without a failure. Finally, every one of its input ports has to be ready, i.e., a value has to be available to be consumed from it, either one that was produced during the computation or provided as default. Formally:

Definition 2.6.4 (Enabled processor)**.** Given a Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ and its local state $ls = (Iv, Ov, ipv, opv, es, ii)$, a processor $p \in P$ is said to be *enabled* iff it holds that:

(i) $es(p) =$ "scheduled", and

(ii) for every control edge $(p', p) \in E_c$, $es(p') =$ "finished", and

(iii) each input port of p is ready in ls.

Notice that during one Scufl graph run each processor at the top level can start processing of the input at most once, so it can produce at most one result value and thus each data edge transports at most one value.

[5]Although, in the intermediary processor case the ordering of the elements of the result list would be always the same and not correspond to order in which the input values have arrived.

2.6.3 Finished Scufl graphs

Here we explain when a Scufl graph is considered to be *finished*. Informally, a Scufl graph is finished when all the workflow input values were propagated, all values on processor output ports were propagated and there are no more processors that can start preparing, are preparing or are waiting.

Definition 2.6.5 (Finished Scufl graph)**.** A Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ is said to be *finished* in a local state $ls = (Iv, Ov, ipv, opv, es, ii)$ iff it holds that:

(i) for every workflow input $i \in I$ it holds that $Iv(i) = \perp$,

(ii) for every processor output port $(p, l) \in \pi_o$ it holds that $opv(p, l) = \perp$,

(iii) none of the scheduled processors is enabled, and

(iv) there are no preparing or waiting processors.

Furthermore, we say that the Scufl graph *finished with a success*, when its every workflow output $o \in O$ is ready, otherwise we say it *finished with a failure*.

This definition of finishing with a success or failure is implied by the fact that a Scufl graph can be nested and thus must produce values for its every workflow output, so that the processor in which it is nested can produce values on its every output port. However, in the real Taverna two exceptions are present which we briefly discuss here. First, for a Scufl graph that is not nested, i.e., the top level Scufl graph, it is enough to have at least one of its workflow outputs ready so that it finishes with a success. Second, a nested Scufl graph that iterates, i.e., was executed for a nested value in the value computed by the product strategy, always finishes with a success, even if none of its workflow outputs are ready. In the result of such iteration the empty string is used to fill in the missing results for workflow outputs that were not ready, but only when in a subsequent iteration step this workflow output becomes ready. However, if during all iterations a nested Scufl graph has not produced any value on a certain port, then the associated nested processor will fail anyway. For example, assume a nested Scufl graph with one workflow input and one workflow output is defined such that it returns its input value when it is unequal to "x", and no value otherwise. Then, if it iterates over ["x", "y", "x", "y", "x"] it returns ["", "y", "", "y"]. However, iterating with such

a nested Scufl graph over a list with just "x" elements causes a failure of the nested processor.

The inclusion of the extra empty values seems an attempt to save such iterating nested Scufl graphs from failure. However, the empty values will probably be misinterpreted in the remaining part of the Scufl graph in which the iteration over the nested Scufl graph occurred. Moreover, the absence of the extra empty values when they are not followed by ordinary results, may also confuse the user. For example, consider the Scufl graph in Fig. 2.4, where the nested Scufl graph is used to submit a paper to a PhD symposium and apply for a grant to visit it, and the "Declaration_of_expenses" processor has the dot product strategy specified. Let us assume, that this Scufl graph is started with a list of three PhD students {"X", "Y", "Z"} and during the iteration the papers written by "X" and "Z" are accepted, but for some formal reasons they do not get grants and the paper written by "Y" is rejected, but he gets a grant anyway since the money are available. That is the list {"", "gnY"}, where "gnY" is the grant number for "Y" is returned on the output port *grant_number*, and the list {"idX", "", "idZ"}, where "idX" is the accepted paper identifier for "X" while "idZ" for "Z", is returned on the output port $conference\_name$. Now, if the "Declaration_of_expenses" processor is not prepared to handle empty values, "X" will have his expenses refunded even though he had no grant, "Y" will get money from his grant even though he did not go to the symposium and "Z" will not have his expenses refunded, despite the fact that he was in the same situation as "X". Although the first thing is not bad in this context, the remaining two probably are. Furthermore, if the symposium chair was used to running this Scufl graph for individual PhD students, he would probably be dissatisfied by the different behavior, i.e., running this Scufl graph separately for any of "X", "Y" and "Z" would alert him with an error.

Therefore we have chosen not to allow in our formal semantics Taverna's exception for nested Scufl graphs and define them also to be finished with failure if not all their workflow outputs have produced a value. For uniformity we also do not allow Taverna's exception for the top level Scufl graph, so also there we define the notion such that all workflow outputs must produce a result in order for it to finish with success.

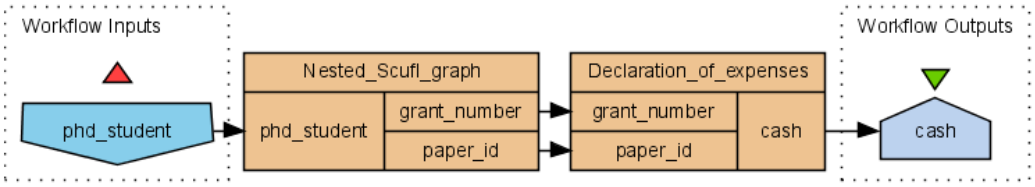

Figure 2.4: Iteration over a nested Scufl graph

2.6.4 Scufl graph initialization and result collection

When a Scufl graph starts execution its state needs to be reset such that any remaining state properties such as intermediate and final results of the previous execution are removed. Therefore we introduce the notion of an *initial state* in which we reset the workflow outputs, the processor input ports, the processor output ports, the processor states and the iteration indices. Note that the workflow inputs are not required to be empty. Formally the notion is defined as follows.

Definition 2.6.6 (Initial state)**.** A local state $ls = (Iv, Ov, ipv, opv, es, ii)$ of Scufl graph $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$ is said to be an *initial state* iff:

(i) $Ov = \{(o, \bot) \mid o \in O\}$,

(ii) $ipv = \{(p_{in}, \bot) \mid p_{in} \in \pi_i\}$,

(iii) $opv = \{(p_{out}, \bot) \mid p_{out} \in \pi_o\}$,

(iv) $es = \{(p, \text{“scheduled”}) \mid p \in P\}$, and

(v) $ii = \{(p, \epsilon) \mid p \in P\}$.

An initial state for which $Iv^{-1}(\{\bot\}) = \emptyset$, with $Iv^{-1}(X) := \{i \mid Iv(i) \in X\}$, is called *full*, if $Iv^{-1}(\{\bot\}) = I$ it is called *clean* and otherwise the initial state is called *partial*.

In addition we define the function init to return the initial local state of a given Scufl graph after initiating its workflow inputs with values stored on fields of a given tuple. Formally, the function $\mathsf{init} : \mathcal{G} \times \mathcal{V}_{tup} \rightarrow LS$ is defined as a partial function such that for a Scufl graph g and a tuple t where $\mathrm{dom}(t) \subseteq I$ it holds that $\mathsf{init}(g, t)$ is the initial state $(t \cup \bar{t}, Ov, ipv, opv, es, ii)$ of g, where

$\bar{t} = \{(i, \bot) \mid i \in (I \setminus \mathrm{dom}(t))\}$. Note that the returned initial state is full if $\mathrm{dom}(t) = I$.

We also define the function result to return the tuple of values computed on the workflow outputs in a given local state of a given Scufl graph. Formally, the partial function $\mathsf{result} : LS \to \mathcal{V}_{tup}$ is defined such that $\mathsf{result}(ls) = Ov$ if $ls = (Iv, Ov, ipv, opv, es, ii)$ and $Ov \in \mathcal{V}_{tup}$. Observe that $\mathsf{result}(ls)$ is defined if a Scufl graph g finished with a success in local state ls.

2.6.5 State transitions

In this section we describe the possible transitions of the state of a Scufl graph. Recall that a system and its state is defined by a hierarchical Scufl graph g and a global state gs of g. For each type of transition we will specify a precondition over gs that must be satisfied and specify the new global state gs' such that the transition $gs \rightsquigarrow gs'$ is possible.

Before we proceed with the full description of the transitions, we summarize them in a brief and informal overview:

Propagation of values from workflow inputs (PROPWI) The values in the workflow inputs are propagated to the processor input ports and workflow outputs to which they are connected by data edges. At their destination they are added to any value that is already present there according to the incoming-links strategy.

Initializing processor execution (INITPE) A scheduled and enabled processor is prepared for execution, i.e., the output port values are initialized and the iteration index is set to the first suitable value in the result computed by the product strategy.

Starting a service call by a basic processor (STARTSC) A call is made to the service associated with the basic processor, with the value indicated by the iteration index as a parameter.

Finishing successfully a service call by a basic processor (SUCFSC) A call to a service succeeds and returns a value. The value is distributed and inserted into the different output port values of the processor. The iteration index is moved to the next suitable value.

Failure of a service call by a basic processor (FAILSC) A call to a service fails and so the whole execution of the processor fails.

Starting a nested Scufl graph execution (STARTNSGE) The nested Scufl graph is initialized with the value indicated by the iteration index.

Finishing successfully a nested Scufl graph execution (SUCFNSGE) The nested Scufl graph finishes with success and returns a value. This value is distributed and inserted into the different output port values of the processor. The iteration index is moved to the next suitable value.

Failure of a nested Scufl graph execution (FAILNSGE) The nested Scufl graph finishes with failure, and so the whole execution of the processor fails.

Finishing processor execution (FINPE) If the iteration index is undefined because there is no next suitable value, the executing of the processor finishes with a success.

Propagation of values from processor output ports (PROPOP) If a processor is finished, but not failed, the values of its output ports are propagated to the processor input ports and workflow outputs to which they are connected by data edges. At their destination they are added to any value that is already there according to the specified incoming-links strategy.

We now describe the transitions in full detail using the following notation. For a local state $ls = (Iv, Ov, ipv, opv, es, ii)$ we let $ls[\mathbf{Iv} := Iv']$ denote the local state $(Iv', Ov, ipv, opv, es, ii)$. In a similar fashion we define $ls[\mathbf{Ov} := Ov']$, $ls[\mathbf{ipv} := ipv']$, $ls[\mathbf{opv} := opv']$, $ls[\mathbf{es} := es']$ and $ls[\mathbf{ii} := ii']$, as the local states equal to ls but with the indicated tuple position replaced with the new value. For a function f and values x and y, we let $f[x \mapsto y]$ denote the function that is equal to f except that it maps x to y, i.e., the function $\{(x', y') \mid (x', y') \in f, x' \neq x\} \cup \{(x, y)\}$. For two functions f and h, we let $f[h]$ denote the function equal of f except for values x for which h is defined, which are mapped to $h(x)$, i.e., the function $\{(x', y') \in f \mid \neg \exists y'' : (x', y'') \in h\} \cup h$.

Propagation of values from workflow inputs (PROPWI)

Consider a workflow input $i \in I$. If the value of i is defined, i.e., $Iv(i) \neq \bot$, then this value is removed from the workflow input and added to the input ports and workflow outputs to which i is connected with a data edge. For

each such input port and workflow output the data is added as specified by the corresponding incoming-links strategy. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$i \in I$, $Iv(i) \neq \bot$

transition: $gs \rightsquigarrow gs[g \mapsto ls']$ where
$ls' = ls[\mathbf{Iv} := Iv[i \mapsto \bot]][\mathbf{Ov} := Ov[Ov']][\mathbf{ipv} := ipv[ipv']]$, with
$Ov' = \{(o, [\![ils(o)]\!](Ov(o), Iv(i)) \mid o \in O, (i, o) \in E_d\}$ and
$ipv' = \{(p_{in}, [\![ils(p_{in})]\!](ipv(p_{in}), Iv(i))) \mid p_{in} \in \pi_i, (i, p_{in}) \in E_d\}$

Note that if a workflow input has no outgoing data edges, its value is anyway reset to $\bot$.

Initializing processor execution (INITPE)

Consider an enabled processor $p \in P$ in state "scheduled" and let v be the value computed by the product strategy of p from its available input port values and default values, i.e., $v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$ where t_1 is the tuple constructed from the available values on the input ports and t_2 is the tuple constructed from the default values for the input ports for which no value is available, i.e., $t_1 = \{(l, ipv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) \in \mathcal{V}_{tav}\}$ and $t_2 = \{(l, dv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) = \bot\}$. The output port values of the output ports of the processor p are initialized with an empty list, the iteration index of p is set to the first iteration value in v, and the state of the processor is set to "preparing". Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, p is enabled in ls, $es(p) =$ "scheduled",
$t_1 = \{(l, ipv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) = \bot\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$

transition: $gs \rightsquigarrow gs[g \mapsto ls']$ where
$ls' = ls[\mathbf{opv} := opv[opv']][\mathbf{es} := es[p \mapsto$ "preparing"$]][\mathbf{ii} := ii[p \mapsto \mathsf{first}(v)]]$, with $opv' = \{((p, l), []) \mid (p, l) \in \pi_o\}$

Starting a service call by a basic processor (STARTSC)

Consider preparing basic processor $p \in P$ and let v again be the value computed by the product strategy of p from its input port values. The precondition is that in v there is a next iteration element, i.e., $ii(p) \in \mathcal{I}$, and that the service was not yet called for this element, i.e., $es(p) =$ "preparing", then the execution state of p is set to "waiting". This models the real world event that the service $\lambda(p)$ is called with the parameters $\mathsf{get}(v, ii(p))$. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in TS$, $es(p) =$ "preparing", $ii(p) \in \mathcal{I}$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto \text{"waiting"}]]]$

Finishing successfully a service call by a basic processor (SUCFSC)

Consider a basic processor $p \in P$ that is waiting for the result of a service call, i.e., $es(p) =$ "waiting", and let v again be the value computed by the product strategy of p from its input port values. For a possible result of such a service call the respective fields are inserted into the output port values at the position indicated by the iteration index, the execution state is set to "preparing" and the iteration index is advanced one position. This models the real world event that the previously made service call succeeds and returns a certain value. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in TS$, $es(p) =$ "waiting",
$t_1 = \{(l, ipv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) = \perp\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$, $(\mathsf{get}(v, ii(p)), t) \in [\![\lambda(p)]\!]$,

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{opv} := opv[opv']][\mathbf{es} := es'][\mathbf{ii} := ii']]$ where
$opv' = \{((p, l), \mathsf{put}(opv(p, l), ii(p), t(l))) \mid (p, l) \in \pi_o\}$,
$es' = es[p \mapsto \text{"preparing"}]$ and
$ii' = ii[p \mapsto \mathsf{next}(v, ii(p))]$

Failure of a service call by a basic processor (FAILSC)

Consider a basic processor $p \in P$. If the processor is waiting for the result of a call, i.e., $es(p) =$ "waiting", then the call might fail and its execution state becomes "failed". This models the real world event that the call to the service $\lambda(p)$ failed. This leads to the following formal specification of the transition:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in TS$, $es(p) =$ "waiting"

transition: $gs \rightsquigarrow gs[g \mapsto ls[\textbf{es} := es[p \mapsto \text{"failed"}]]]$

Starting a nested Scufl graph execution (STARTNSGE)

This transition is very similar to the starting of a service call by a basic processor, except that the processor is not a basic processor but a nested Scufl graph and rather than starting a call to a service the nested Scufl graph $\lambda(p)$ is initialized for this iteration element. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in \mathcal{G}$, $es(p) =$ "preparing", $ii(p) \in \mathcal{I}$,
$t_1 = \{(l, ipv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p, l)) \mid (p, l) \in \pi_i, ipv(p, l) = \perp\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\textbf{es} := es']][\lambda(p) \mapsto \mathsf{init}(\lambda(p), \mathsf{get}(v, ii(p)))]$
where
$es' = es[p \mapsto \text{"waiting"}]$

Finishing successfully a nested Scufl graph execution (SUCFNSGE)

This transition is very similar to the finishing successfully of a service call by a basic processor, except that the processor is not a basic processor but a nested Scufl graph and it is required in the precondition that the nested Scufl graph $\lambda(p)$ must have finished with success in $gs(\lambda(p))$, and the result tuple is composed from the output ports of the nested Scufl graph, i.e., $t = \mathsf{result}(gs(\lambda(p)))$. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in \mathcal{G}$, $es(p) =$ "waiting",
$\lambda(p)$ finished with a success in $gs(\lambda(p))$,
$t_1 = \{(l, ipv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) \in \mathcal{V}_{tav}\}$,
$t_2 = \{(l, dv(p,l)) \mid (p,l) \in \pi_i, ipv(p,l) = \bot\}$,
$v = [\![ps(p)]\!]^{type_i(p)}(t_1 \cup t_2)$, $t = \mathsf{result}(gs(\lambda(p)))$,

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{opv} := opv[opv']][\mathbf{es} := es'][\mathbf{ii} := ii']]$ where
$opv' = \{((p,l), \mathsf{put}(opv(p,l), ii(p), t(l))) \mid (p,l) \in \pi_o\}$,
$es' = es[p \mapsto$ "preparing"$]$ and
$ii' = ii[p \mapsto \mathsf{next}(v, ii(p))]$

Failure of a nested Scufl graph execution (FAILNSGE)

This transition is very similar to the failure of a service call by a basic processor, except that the processor is not a basic processor but a nested Scufl graph and it is required in the precondition that the nested Scufl graph $\lambda(p)$ must have finished with a failure in its local state $gs(\lambda(p))$. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $\lambda(p) \in \mathcal{G}$, $es(p) =$ "waiting",
$\lambda(p)$ finished with a failure in $gs(\lambda(p))$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto$ "failed"$]]]$

Finishing processor execution (FINPE)

If the processor is preparing and there is no next iteration index, then the state of the processor becomes "finished". Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$p \in P$, $es(p) =$ "preparing", $ii(p) = \bot$

transition: $gs \rightsquigarrow gs[g \mapsto ls[\mathbf{es} := es[p \mapsto$ "finished"$]]]$

Propagation of values from processor output ports (PROPOP)

Consider a processor output port $(p, l) \in \pi_o$. If the value of (p, l) is defined, i.e., $opv(p, l) \neq \bot$ and the processor is finished, but not failed, then this value is removed from the processor output port and added to the input ports and workflow outputs to which output port (p, l) is connected with a data edge. For each such input port and workflow output the data is added as specified by the corresponding incoming-links strategy. Formally:

precondition: $g = (I, O, P, \pi_i, \pi_o, E_d, E_c, \lambda, ils, ps, dv)$,
$gs(g) = ls = (Iv, Ov, ipv, opv, es, ii)$,
$(p, l) \in \pi_o$, $opv(p, l) \neq \bot$, $es(p) =$ "finished"

transition: $gs \rightsquigarrow gs[g \mapsto ls']$ where
$ls' = ls[\mathbf{Ov} := Ov[Ov']][\mathbf{ipv} := ipv[ipv']][\mathbf{opv} := opv[(p, l) \mapsto \bot]]$, with
$Ov' = \{(o, [\![ils(o)]\!](Ov(o), opv(p, l)) \mid o \in O, ((p, l), o) \in E_d\}$ and
$ipv' = \{(p_{in}, [\![ils(p_{in})]\!](ipv(p_{in}), opv(p, l))) \mid p_{in} \in \pi_i, ((p, l), p_{in}) \in E_d\}$

Note that if a processor output port has no outgoing edges, its value is anyway reset to $\bot$.

Scufl graph run

The specification of possible transitions defines a transition system that can be used to describe the semantics of Scufl graphs. An instance of a computation of a particular Scufl graph g, i.e., a sequence of successive global states reached during the computation, will be called a *run*[6]. We will denote a run of global states $gs_1, \ldots, gs_n$ of g as $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n$, by which we mean that $gs_i \rightsquigarrow gs_{i+1}$ for each $1 \leq i \leq n - 1$.

A run $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n$ of g will be called a *cleanly initialized run* if it starts with an initial local state of g, i.e., $gs_1(g)$ is initial, and clean initial local states of all the nested graphs, i.e., for all $g' \in \mathcal{G}_g$ such that $g' \neq g$ the local state $gs_1(g')$ is a clean initial state.

2.6.6 Soundness of the transition system

To check the completeness of our semantics definition we are going to formally prove a property of Scufl graphs, which states that for every Scufl graph g all

[6]In the next chapter we will also use the term *run* in Definition 3.7.8, yet it will be in a different context and the second definition does not cause ambiguity.

its cleanly initialized runs that start with g initialized with any input values of any type[7] and possibly missing input values eventually finish, either with success or with failure.

At the same time this exercise shows that the formal semantics as defined in this thesis can be used in proofs of this kind.

Theorem 2.6.7. *For every Scufl graph g and any of its cleanly initialized runs $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n$:*

(i) there is a maximum number of steps that this run can be extended with, i.e., such $m \in \mathbb{N}$ that for every run $gs_1 \rightsquigarrow \ldots \rightsquigarrow gs_n \rightsquigarrow gs_{n+1} \rightsquigarrow \ldots \rightsquigarrow gs_k$ of g it holds that $k \leq m$, and

(ii) if in gs_n none of the transitions is possible then g is finished.

Proof of Theorem 2.6.7 In the following we assume g to be a Scufl graph and gs its global state.

We first show that the runs are of finite length. The idea is to show that the global state of g in some sense decreases with each transition and this decreasing cannot proceed indefinitely. For that we define a *global state vector* which is a natural number vector. The composition of the vector is based on the properties of combined local states of the Scufl graphs that occupy the same level of the tree given by the nesting graph $\mathcal{N}_g$ (in the following referred to as the nesting tree)[8], i.e., graphs that as the nodes of the tree have the same depth. Let $\mathcal{N}_g(k)$ be the set of graphs at depth k of the nesting tree $\mathcal{N}_g$, i.e., $\mathcal{N}_g(0) = \{g\}$ and $\mathcal{N}_g(k+1) = \{g' \mid (g, p, g') \in E, g \in \mathcal{N}_g(k)\}$. For each non-empty level k of the nesting tree the vector contains six subsequent properties: (1) the total number of workflow inputs of graphs $g_k \in \mathcal{N}_g(k)$ which in $gs(g_k)$ are not empty, (2) the total number of processors of graphs $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are scheduled, (3) the total number of processors of graphs $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are neither finished nor failed, (4) the total number of elements that still have to be iterated by processors of graphs $g_k \in \mathcal{N}_g(k)$ in $gs(g_k)$, (5) the total number of processors in graphs $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are preparing (6) the total number of processor output ports in $g_k \in \mathcal{N}_g(k)$ that in $gs(g_k)$ are empty.

[7] Under our liberal type semantics this includes heterogeneous values, and therefore all complex values.

[8] Recall (see Section 2.4.1) that we assume Scufl graphs to be hierarchically nested and thus the nesting graph $\mathcal{N}_g$ is a tree.

The following functions of signature $\mathbb{N} \to \mathbb{N}$ give the values of those properties. In their definition we assume that $g^k = (I^k, O^k, P^k, \pi_i^k, \pi_o^k, E_d^k, E_c^k, \lambda^k, ils^k, ps^k, dv^k)$ and $gs(g^k) = (Iv^k, Ov^k, ipv^k, opv^k, es^k, ii^k)$.

1. $\mathsf{notewfi}_g^{gs}(k) = \big|\{\, i \mid i \in I^k, Iv^k(i) \neq \bot, g^k \in \mathcal{N}_g(k) \,\}\big|$
2. $\mathsf{psched}_g^{gs}(k) = \big|\{\, p \mid p \in P^k, es^k(p) = \text{“scheduled”}, g^k \in \mathcal{N}_g(k) \,\}\big|$
3. $\mathsf{pnotff}_g^{gs}(k) =$
   $\big|\{\, p \mid p \in P^k, es^k(p) \neq \text{“finished”}, es^k(p) \neq \text{“failed”}, g^k \in \mathcal{N}_g(k) \,\}\big|$
4. $\mathsf{iterleft}_g^{gs}(k) = \sum\{\, \mathsf{togo}(v, i) \mid p \in P^k, ii^k(p) = i, g^k \in \mathcal{N}_g(k) \,\}$
5. $\mathsf{pprep}_g^{gs}(k) = \big|\{\, p \mid p \in P^k, es^k(p) = \text{“preparing”}, g^k \in \mathcal{N}_g(k) \,\}\big|$
6. $\mathsf{ewfo}_g^{gs}(k) = \big|\{\, o \mid o \in O^k, Ov^k(i) = \bot, g^k \in \mathcal{N}_g(k) \,\}\big|$

where $\mathsf{togo}(v, i) : \mathcal{V}_{ext} \times \mathcal{I} \to (\mathcal{V}_{ext} \cup \bot)$ is defined such that $\mathsf{togo}(v, i) = 0$ if $\mathsf{get}(v, i) = \bot$ and $\mathsf{togo}(v, i) = 1 + \mathsf{togo}(v, i)$ if $\mathsf{get}(v, i) \neq \bot$. It is easy to see that the functions are well defined in any state of a cleanly initialized run.

In the vector the components corresponding to smaller depths in the nesting tree precede the ones for bigger depths. Formally the vector is defined as follows:

$$(\mathsf{notewfi}_g^{gs}(0), \mathsf{psched}_g^{gs}(0), \mathsf{pnotff}_g^{gs}(0), \mathsf{iterleft}_g^{gs}(0), \mathsf{pprep}_g^{gs}(0), \mathsf{ewfo}_g^{gs}(0)$$
$$\ldots$$
$$\mathsf{notewfi}_g^{gs}(h_g), \mathsf{psched}_g^{gs}(h_g), \mathsf{pnotff}_g^{gs}(h_g), \mathsf{iterleft}_g^{gs}(h_g), \mathsf{pprep}_g^{gs}(h_g), \mathsf{ewfo}_g^{gs}(h_g))$$

where h_g is the height of the nesting tree, i.e., the biggest number k such that $\mathcal{N}_g(k) \neq \emptyset$. Observe that the vector is thus of finite size determined by the height of the nesting tree, i.e., its size is equal six times the height of the nesting tree.

We are now going to show that under a lexicographical ordering each transition in a cleanly initialized run decreases the global state vector. For that we are going to list how each state transition changes the vector:

PROPWI does not increase any component and decreases the number of not empty workflow inputs (1),

INITPE increases the number of remaining iterations (4) and the number preparing processors (5), but at the same time decreases the number of scheduled processors (2),

STARTSC does not increase any component and decreases the number of preparing processors (5),

SUCFSC increases the number of preparing processors (5), but at the same time decreases the number of remaining iterations (4),

FAILSC does not increase any component and decreases the number of processors that are neither finished nor failed (3),

STARTNSGE increases the part of the vector that corresponds to the nested Scufl graph, which is on a bigger depth thus less important in our ordering, and decreases the number of scheduled processors (2),

SUCFNSGE similarly as SUCFSC increases the number of preparing processors (5), but at the same time decreases the number of remaining iterations (4),

FAILNSGE does not increase any component and decreases the number of processors that are neither finished nor failed (3),

FINPE does not increase any component and decreases the number of processors that are neither finished nor failed (3),

PROPOP decreases the number of empty workflow outputs (6).

It is well known from set theory that the set of natural number vectors of a given length with lexicographical ordering is a well-founded partially ordered set and thus it does not contain an infinite descending chain. For the self containment of this work we show this formally in the Appendix A (see Corollary A.0.7). This proves that all runs are of finite length, since from the non-existence of an infinite descending chain it follows that there is a bound on the number of transitions by which a given run can be extended.

To complete the proof of Theorem 2.6.7 we are going to show that if a state has been reached in which no transitions are possible, i.e., none of the transitions has its preconditions satisfied, then g is finished. The proof will follow by induction on the height of the nesting tree $\mathcal{N}_g$

We first assume that the nesting tree $\mathcal{N}_g$ is of height 1 and that in a global state gs' of g none of the transitions has its preconditions satisfied. We will show that g is finished in $gs'(g)$. For that we look at the four conditions in definition 2.6.5. It is clear that (i) directly follows from the unfulfillment of the preconditions for transition PROPWI, (ii) directly follows from the unfulfillment of the preconditions for transition PROPOP and (iii) directly follows from the unfulfillment of the preconditions for transition INITPE. As for (iv) let us first notice that if the nesting tree $\mathcal{N}_g$ is of height 1, then g contains only basic processors, i.e., $\mathcal{N}_g = \{\{g\}, \emptyset\}$. If there would be any preparing processor, then either STARTSC or FINPE transitions would be possible depending on whether there is a next iteration element for that processor. Also, if there would be any waiting processor, then both the SUCFSC and FAILSC transitions would be possible[9]. Thus all the conditions for a finished Scufl graph are satisfied.

We now assume that the thesis holds for all the Scufl graphs with the nesting tree of height smaller or equal to n and we are going to show that it also holds for all graphs with the nesting tree of height $n+1$. Let the nesting tree $\mathcal{N}_g$ be of height $n+1$ and let gs' be a global state g such that none of the transitions has its preconditions satisfied. We will show that g is finished in $gs'(g)$. As before we look at the conditions in the definition 2.6.5. For conditions (i), (ii) and (iii) the reasoning follows. As for (iv) the argument for non-existence of preparing processors remains the same. Similarly for the non-existence of waiting basic processors. The only thing left to show is that there are no waiting processors that represent nested Scufl graphs. Let us assume by contradiction that a waiting processor p exists in g and represents a nested Scufl graph. Because preconditions for SUCFNSGE and FAILNSGE transitions are not satisfied, then the nested Scufl graph of p cannot be finished. Yet, the nesting tree of that nested Scufl graph is of height smaller or equal to n and since we have assumed that no transitions are possible, it follows from the induction assumption that the nested Scufl graph is finished. This completes the proof by contradiction and thus the proof by induction. □

It is easy to see that a cleanly initialized run of a Scufl graph may finish with failure if (1) not all input port values are available or if (2) a basic processor fails. In both cases some processors can never produce their output

[9]Recall that we do not include the state of the external services in our formal model and thus there is no dependency on any such state in the preconditions for the transitions.

which may prevent some or all workflow outputs from becoming ready. It can also be observed that (1) and (2) are the only reasons for a Scufl graph not to succeed and thus for every Scufl graph g all its cleanly initialized runs that start with g in a fully initialized local state, i.e., with all the workflow input values present, eventually terminate with success if we exclude the failure of transitions. Although we do not give here a formal proof, this follows intuitively from the facts that processors without input ports are immediately enabled, all processor input ports have either incoming data edges or a default value specified and because Scufl graphs contain no cycles.

2.7 Dealing with heterogeneous values in Taverna

Until now in the discussion of the semantics of Scufl we focused our attention on homogeneous lists. Yet, heterogeneous values can be created in Taverna with the use of the merge incoming-links strategy or by an iteration on a processor that returns values with various nesting depths in its subsequent executions, e.g., sometimes lists and sometimes lists of list. Unfortunately, heterogeneous values are not processed consistently in the current implementation of Scufl. In Section 2.2 we gave an example that some services, i.e., the built-in flatten operation, are not prepared to handle such values. Furthermore, the way the dot product is implemented in Taverna yields sometimes rather unexpected results for heterogeneous values. Informally, the dot product in Taverna is computed by iterating over the tuples in both arguments. During the iteration, the subsequent pairs of tuples are combined, i.e., first with first, second with second and so on. The combinations are placed in the result nested list on positions pointed by the longer of the indexes of the combined tuples. If both indexes have the same length, the left one is chosen. This is different from our definition of the $\cdot$ operator from Section 2.5.4 because we structure the result according to the argument with the higher nesting depth, which means that the indexes for tuples combinations are taken from an argument chosen in advance and not determined for each combination of tuples separately. Of course, for homogeneous values both methods produce the same results, since for homogeneous list all its tuple indexes have the same length. Yet, in Taverna the resulting indexing can contain gaps. For example, if a, b, c, x, y and z are tuples, and a dot product

of $[[[a,b]],c]$ and $[[x,y,z]]$ is computed, then the result would contain: $a \cup x$ on index 1/1/1, $b \cup y$ on index 1/1/2 and $c \cup z$ on index 1/3. The 1/2 position would have to be filled up by some kind of empty value. This problem does not occur if the definitions of dot product from Section 2.5.4 are taken, i.e., $[[[a,b]],c] \cdot [[x,y,z]] = [[[a \cup x, b \cup y]], c \cup z]$ and $[[[a,b]],c] \cdot_r [[x,y,z]] = [[[a \cup x]]]$.

Here we discuss two possible solutions to the heterogeneous values problem in Taverna. One, is to adopt the formal semantics from this thesis which seems intuitive while at the same time allows heterogeneous values everywhere and deals with them consistently. We elaborate on this in Section 2.7.1. The other solution is to avoid heterogeneous values at all, which we discuss in further detail in Section 2.7.2.

2.7.1 Allowing heterogeneous lists

The semantics defined in this thesis deals with heterogeneous lists intuitively and consistently, yet its adoption in the workbench may require additional effort for adjusting some of the services. For examples, for the built-in flatten operation a definition is possible that processes the heterogeneous values consistently, that is:

$$\mathsf{flat}(x) \quad = \quad \begin{cases} [] & \text{if } x = [] \\ list(x_1) + \ldots + list(x_n) & \text{if } x = [x_1, \ldots, x_n] \end{cases}$$

where $list(x) = x$ for list values and $list(x) = [x]$ for mime values. Note, that with this definition, flattening of $[[x],[[y]]]$ yields $[x,[y]]$ and not $[[x],[y]]$ as it is the case in Taverna.

It is also possible to extend the type coercion mechanism described in Section 2.2. If a certain service that requires its input lists to be homogeneous and of a specific nesting depth, gets a value that is non homogeneous or is of lower nesting depth, then there is always an intuitive interpretation of subvalues in that value as more deeply nested ones, namely by nesting them in singleton lists. For this a homogenisation function $\mathsf{hom}_\tau : [\![\tau]\!] \to [\![\tau]\!]$ can be used, that maps all complex values of type τ to homogeneous complex value with the maximum nesting depth possible in τ. It is defined such that:

- $\mathsf{hom}_{\mathcal{M}}(x) = x$,
- $\mathsf{hom}_{[\tau]}(x) = [\mathsf{hom}_\tau(x)]$, if $x \in [\![\tau]\!]$, and

- $\mathsf{hom}_{[\tau]}([x_1, \ldots, x_n]) = [\mathsf{hom}_\tau(x_1), \ldots, \mathsf{hom}_\tau(x_n)]$, if $[x_1, \ldots, x_n] \notin [\![\tau]\!]$.

The function hom_τ packs values that do not have the maximum nesting depth allowed in τ into singleton lists, and if the value does have the right nesting depth and $\tau = [\sigma]$ then it applies itself to the elements of the list for the type σ. For example, $\mathsf{hom}_{[[[\mathcal{M}]]]}([1,[2]]) = [\mathsf{hom}_{[[\mathcal{M}]]}([1,[2]])] = [[\mathsf{hom}_{[\mathcal{M}]}(1), \mathsf{hom}_{[\mathcal{M}]}([2])]] = [[[\mathsf{hom}_{\mathcal{M}}(1)], [\mathsf{hom}_{\mathcal{M}}(2)]]] = [[[1],[2]]]$. Thanks to this function we can safely assume that all services can deal with all homogeneous and heterogeneous complex values that belong to their input type.

It is worth pointing out that the existence of such type coercion is consistent with Taverna's philosophy of trying to fix the type mismatches for the user.

2.7.2 Adapting the semantics to avoid heterogeneous lists

Assuming that the heterogeneous values cannot be provided by the user, i.e., as workflow inputs or default values, to avoid them always, we have to make sure that they cannot be obtained during the computation. New values appear in a Scufl graph in the following cases: (1) they are produced in a processor execution, (2) they are created in the incoming-links strategy computation, (3) they are created in the product strategy computation, or (4) they are produced in a processor iteration.

As for (1), a service that was provided with only homogeneous values as arguments could produce heterogeneous results. One possibility is to interpret this as a failure, another is to always adapt the result value with the homogenisation function, i.e., if the service call returns v, then use $\mathsf{hom}_\tau(v)$ as the result, where τ is the smallest type of v.

As for (2), the select-first strategy does not change the values, so it cannot cause a heterogeneous value to appear. Yet, the merge strategy can, if the subsequent values provided to it are of different nesting depth. Similarly as in (1), this can be remedied with the use of the homogenisation function to extend the merge function as follows:

$$[\![\mathsf{merge}_{hom}]\!](t, v) = \begin{cases} [v] & \text{if } t = \bot \\ \mathsf{hom}_{min(\{\tau \mid (t+[v]) \in [\![\tau]\!]\})}(t + [v]) & \text{otherwise} \end{cases}$$

where $min : \mathcal{P}(\mathcal{T}_{tav}) \to \mathcal{T}_{tav}$ returns the minimal type in a set of types.

As for (3), neither the dot nor the cross product can produce heterogeneous values from homogeneous arguments, regardless which definition is chosen, so no extra care is necessary.

Finally, as for (4), a heterogeneous value can be created, if the processor returns results with different nesting depths in the subsequent iteration steps. Again, this can be solved with the use of homogenisation function, this time to extend the put function such that:

$$\mathsf{put}_{hom}(v,\alpha,t) = \begin{cases} \mathsf{put}(v,\alpha,t) & \text{if } \mathsf{put}(v,\alpha,t) = \bot \\ \mathsf{hom}_{min(\{\tau \mid \mathsf{put}(v,\alpha,t) \in [\![\tau]\!]\})}(\mathsf{put}(v,\alpha,t)) & \text{otherwise} \end{cases}$$

2.8 Related work on Scufl semantics

Here we compare the presented work on Scufl semantics with that of Turi et al. in [60]. In that work a calculus is defined to represent Scufl graphs and a semantics is defined for them in terms of function that map workflow input values to a workflow output value.

The most important difference is that in our work we assume that calls to services have side effects, or, in other words, are observable events that are part of the behavior of the system. This means that two computations that call services in a different order or a different number of times, are not considered as equivalent, even if they compute the same output value. Therefore we describe the semantics of a Scufl graph not in terms of functions, but in terms of a transition system that describes which calls are made in which order, which arguments were passed, and which output values are produced in the workflow outputs as the result of the Scufl graph execution. A consequence of the side-effect assumption is that, contrary to what Turi et al. assume, nesting a Scufl graph is no longer a purely syntactic construct because it synchronizes the consumption of values on the input ports and the production of values on the output ports, and so changes the observable behavior of the transition system. The same holds for the control edge, which can only have meaning if the order of computations is an observable aspect of the system.

The second difference with the work by Turi et al. is that their syntax is defined by a statically typed calculus. Since a Scufl graph is polymorphic and can work on inputs of different types, its semantics cannot always be described by a single calculus expression and may require a different one for each

possible type of input value. In addition, the coercion to more deeply nested list types by wrapping and the implicit iteration strategy have to be made explicit in calculus expressions. As a result the mapping of a real Scufl graph, as described in our work, given certain presumed input types for the workflow inputs, to a calculus expression is not simple. In fact, certain peculiarities of Taverna's implicit iteration semantics seem to require operations that are not expressible in the calculus presented by Turi et al. For example, during iteration at deeper nesting levels, certain empty lists are removed. If the identity processor that expects type $\mathcal{M}$ receives the value $[[[]], [[v, w], []], [[]], [[x]], [[]]]$ with $v, w, x \in \mathcal{V}_{\mathcal{M}}$ then it returns $[[], [[v, w]], [], [[x]]]$. Simulating this in the calculus would require a test for empty lists. Another example is the dot product at deeper nesting levels. Assume a processor with two input ports that computes the function $F(x, y)$, expects $\mathcal{M}$ on both its ports, and has the dot product iteration strategy. If it receives the values $v = [[v_1, v_2], [v_3, v_4]]$ and $w = [w_1, w_2, w_3]$ with $v_1, v_2, v_3, v_4, w_1, w_2, w_3 \in \mathcal{V}_{\mathcal{M}}$, then the result is $[[F(v_1, w_1), F(v_2, w_2)], [F(v_3, w_3)]]$. It is possible to compute F for the listed combinations by first flattening v, but the result would then be the flat list $[F(v_1, w_1), F(v_2, w_2), F(v_3, w_3)]$. The difficulty lies in simulating that the result is nested according to the structure of v. So both types of behavior do not seem easily expressible in the calculus unless special operators are added.

The third and last difference is that failure of processors and nested Scufl graphs is not taken into account in the calculus by Turi et al. It can be argued that this aspect should be dealt with at a lower abstraction level, and in Taverna 1 there are indeed other mechanisms such as the specification of the number of retries and alternative services to deal with this. Moreover, in Taverna 2 the concept seems to have been removed entirely from the language level. However, we maintain that it is an interesting and useful feature to have at the language level, for example for specifying powerful fall-back strategies in Scufl itself. It is also essential for understanding the semantics of Scufl in Taverna 1, if only because it is used to represent conditional branching as discussed in Section 2.4.

Chapter 3

DataFlow Language

The complexities of the presented formal semantics of Taverna show that it is desirable to have a cleaner model. In this chapter we propose a new formal language to define COSWs which we call **D**ata**F**low **L**anguage (DFL) [28, 26]. It is a common extension of (1) *Petri nets*, which are responsible for the organization of the processing tasks, and (2) *nested relational calculus*, which is a database query language over complex objects, and is responsible for handling collections of data items, in particular for iteration, and for the typing system. We also show that the existing theoretical results available for the components can be easily adapted for DFL. Finally, we present DFL designer, which is a tool that allows to design, enact and analyze DFL workflows.

Our idea of extending classical Petri nets is not new in general. Colored Petri nets [31] permit tokens to be colored (with finitely many colors), and thus tokens carry some information. In the nets-within-nets paradigm [64] individual tokens have Petri net structure themselves. This way they can represent objects with their own, proper dynamics. Finally, self-modifying nets [63] assume standard tokens, but permit the transitions to consume and produce them in quantities functionally dependent on the occupancies of the places.

To compare, our approach assumes tokens to represent complex data values, which are however static. The transitions are allowed to perform operations on the tokens' contents. Edges can be annotated with conditions and pass only tokens which values satisfy those conditions. There is also a special unnest/nest annotation. When unnest is applied to an output edge of a transition, the output token with a set value is transformed into a set of tokens, one for each element of the set. When nest is applied to an input edge

of a transition, the set of tokens is grouped back into a single "composite" token.

Also the introduction of complex value manipulation into Petri nets was already proposed by others. Oberweis and Sander [43] proposed a formalism called NR/T-nets where places represent nested relations in a database schema and transitions represent operations that can be applied to the database. Although somewhat similar, the purpose of that formalism, i.e., representing the database schema and possible operations on it, is very different from the one presented here. For example, the structure of the Petri net in NR/T-nets does not reflect the workflow, but only which relations are involved in which operations. In our DFL formalism, we can easily integrate external functions and tools as special transitions and use them at arbitrary levels of the data structures. The latter is an important feature for describing and managing COSWs. Therefore we claim that, together with other differences, this makes DFL a better formalism for representing COSWs.

Similarly the idea of using query languages as NRC to model data-centric workflows is also not new. The NRC based BioKleisli system (see 1.2.3) allows to query and transform complex data from heterogeneous sources, including ones available through a network. In our own previous research [21, 22] we have shown that this can be used to express side-effect free COSWs, i.e., COSWs that describe which computations have to be performed, but do not include operations with side effects for which precise specification of control flow is necessary, e.g, is restricted by a certain protocol. In another research of ours we have extended the Taverna workbench (see 1.2.1) to allow defining side-effect free COSWs with the use of XQuery [73, 69], which is a standard hierarchical data query language for XML [72] and is modeled after NRC. Such XQuery based COSWs can be freely intermixed with standard Taverna COSWs, i.e., XQuery based COSWs can use all processors available in Taverna, including processors representing nested Scufl graphs, and can themselves be nested in standard Taverna Scufl graphs. As we notice in [56] for some side-effect free use cases query languages can be successfully used to describe COSWs. In fact, for data-oriented workflows one can use database techniques to define and manage them [2], but this is outside the scope of this thesis.

3.1 Motivation

We start with motivating why the definition of DFL is interesting, even though a few alternative models and notations were already developed for use in COSW systems.

The experiments that are being conducted with the use of COSWs can be quite complex and thus many theoretical problems are interesting. For example, properties of COSWs could be analyzed, similarly as complex business transactions can be automatically checked for liveness, soundness, deadlocks and the like. Furthermore, methods of enactment optimization could be developed for COSWs similarly as for database queries. Yet, the models and notations used in practical COSW systems: (1) sometimes lack formal semantics, which is a necessary first step for development of any formal methods, (2) differ in detail, so the worked up formal methods would have to be adapted to them separately, (3) are designed to be easy in use for COSW creators, but the easiness of formal analysis and verification is not a priority in their construction. Thus the definition of a language with formal semantics that can serve as a comfortable framework for theoretical study and would be a common denominator of languages used in practice, i.e., to which practically applied languages could be mapped, is a worthy goal.

3.2 Combining NRC and Petri nets

3.2.1 Nested relational calculus

The nested relational calculus (NRC) [9] is a query language allowing one to describe functional programs using collection types, e.g., lists, bags, sets, etc. The most important feature of the language is the possibility to iterate over a collection. NRC assumes a set of base types which can be combined to form nested record and collection types. The only collection type we will use are sets.

Besides standard language constructs enabling manipulation of records and sets, NRC contains the three constructs **sng**, **map** and **flatten**. For a value v of a certain type, **sng**(v) yields the singleton set containing v. Operation **map**, applied to a function of type $\tau \to \sigma$, yields a function on sets of type $\{\tau\} \to \{\sigma\}$. Finally, the operation **flatten**, given a set of sets of type τ, yields a flattened set of type τ, by taking the union. These three

basic operations are powerful enough for specifying functions by structural recursion over collections [9].

Similarly as in computational lambda calculus [40], in NRC each nontrivial computation processing a collection yields always a collection again. Even if there is a single final value, it will be returned wrapped up in a collection.

3.2.2 Petri nets

A classical Petri net [41, 51] is a bipartite graph with two types of nodes called *places* and *transitions*. The nodes are connected by directed edges. Only nodes of a different types can be connected. Places are represented by circles and transitions by rectangles.

Definition 3.2.1 (Petri net)**.** A Petri net is a triple $\langle P, T, E\rangle$ where:

- P is a finite set of places,
- T is a finite set of transitions ($P \cap T = \emptyset$),
- $E \subseteq (P \times T) \cup (T \times P)$ is a set of edges.

A place p is called an *input place* of a transition t, if there exists an edge from p to t. A place p is called an *output place* of a transition t, if there exists an edge from t to p. Given a Petri net $\langle P, T, E\rangle$ we will use the following notations:

$$\begin{aligned} \bullet p &= \{t \mid \langle t, p\rangle \in E\} & p\bullet &= \{t \mid \langle p, t\rangle \in E\} \\ \bullet t &= \{p \mid \langle p, t\rangle \in E\} & t\bullet &= \{p \mid \langle t, p\rangle \in E\} \\ \circ p &= \{\langle t, p\rangle \mid \langle t, p\rangle \in E\} & p\circ &= \{\langle p, t\rangle \mid \langle p, t\rangle \in E\} \\ \circ t &= \{\langle p, t\rangle \mid \langle p, t\rangle \in E\} & t\circ &= \{\langle t, p\rangle \mid \langle t, p\rangle \in E\} \end{aligned}$$

and their generalizations for sets:

$$\begin{aligned} \bullet A &= \bigcup_{x \in A} \bullet x & A\bullet &= \bigcup_{x \in A} x\bullet \\ \circ A &= \bigcup_{x \in A} \circ x & A\circ &= \bigcup_{x \in A} x\circ \end{aligned}$$

where $A \subseteq P \cup T$. Places are stores for *tokens*, which are depicted as black dots inside places when describing the run of a Petri net. Edges define the

possible token flow. The semantics of a Petri net is defined as a transition system. A *state* is a distribution of tokens over places. It is often referred to as a *marking* $M : P \rightarrow (\mathbb{N} \cup \{0\})$. The state of a net changes when a transitions *fires*. For a transition t to fire it has to be *enabled*, that is, each of its input places has to contain at least one token. If transition t fires, it *consumes* one token from each of the places in $\bullet t$ and produces one token on each of the places in $t\bullet$.

Petri nets are an established process modeling technique. The interest in them has been constantly growing for the last fifteen years. Many theoretical results are available. One of the better studied classes are *workflow nets*, which are used in workflow management [1].

Definition 3.2.2 (strongly connected)**.** A Petri net is strongly connected if and only if for every two nodes n_1 and n_2 there exists a directed path leading from n_1 to n_2.

Definition 3.2.3 (workflow net)**.** A Petri net $PN = \langle P, T, E \rangle$ is a workflow net if and only if:

(i) PN has two special places: a *source* and a *sink*. The *source* has no input edges, i.e., $\circ source = \emptyset$, and the *sink* has no output edges, i.e., $sink\circ = \emptyset$.

(ii) If we add to PN a transition t^* and two edges $\langle sink, t^* \rangle$, $\langle t^*, source \rangle$, then the resulting Petri net is strongly connected.

3.2.3 How we combine NRC and Petri nets

From NRC we inherit the set of basic operators and the type system. This should make reusing of existing database theory results easy. COSWs specified in DFL, which we further call *dataflows*, could for example undergo an optimization process as database queries do. To deal with the synchronization issues arising from processing of the data by distributed services we will use a Petri-net based formalism which is a clear and simple graphical notation and has an abundance of correctness analysis results. We believe that these techniques can be reused and combined with known results from database theory for verifying the correctness of dataflows.

The fundamental operation in NRC is the map operation **map**. In order to allow a similar kind of iteration in Petri nets we introduce special unnest

and nest edges, that will be distinguished with a star. Unnest edges are outgoing edges of transitions and nest edges are incoming edges. Unnest edges can be used if the function associated with the transition produces a set value. If an outgoing edge is marked as an unnest edge then, if the transition fires, instead of producing in the associated place a single token with the set that is the result of the transition, it will produce a set of tokens, one for each element of the result set. Nest edges can be used if the function associated with the transition requires a set value as a parameter. If an incoming edge is marked as a nest edge then, if the transition fires, instead of consuming from the associated place a single token with a set value, it will consume a set of tokens and combine them into a single set that is used as the parameter of the function.

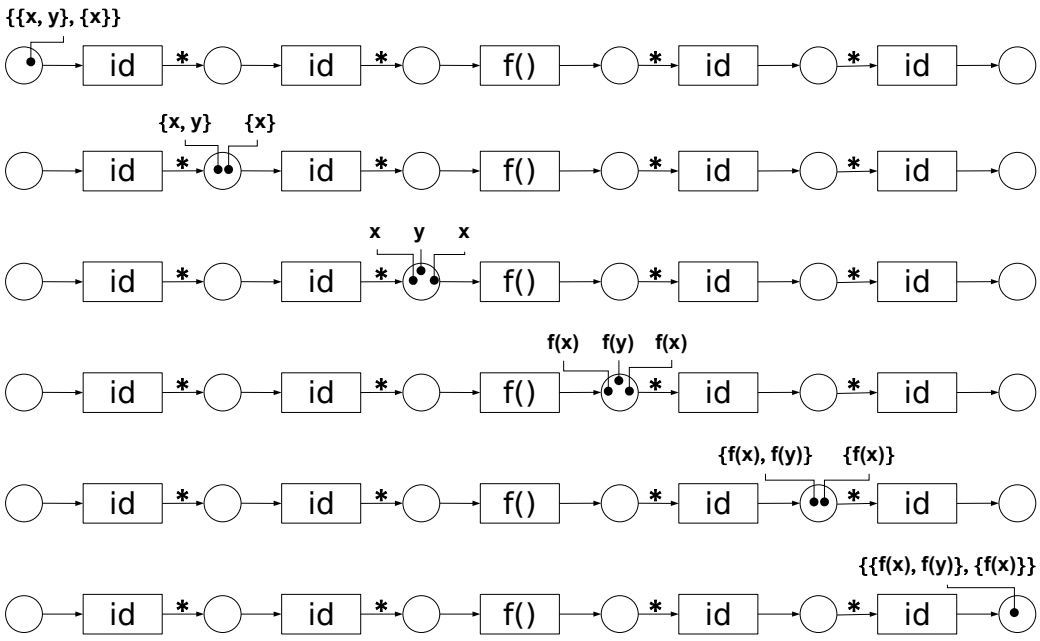

Figure 3.1: Nested iteration example

A simple example with a nested iteration is given in Fig. 3.1. If the dataflow is initiated in the leftmost place with a token representing a set of sets, it will be processed by the identity transition *id* and unnested. Next, the resulting tokens representing sets that were the elements of the input set are unnested themselves by the second pair of identity transition and unnest edge. Finally, function $f()$ is applied to each of the elements of the

unnested subsets and the result is nested twice by two subsequent identity transitions with nest edges. To assure that tokens originating from different sets are not intermixed while nesting and that nesting appears only when all the necessary tokens have arrived, each token carries its unnesting history, which is described in Section 3.5.1.

The unnest and nest edges allow a straightforward representation of the NRC **map** operation in a Petri net formalism and thus make it possible to specify computational processes with the use of this type of structural recursion [58].

3.3 Syntax

We define DFL by starting with Petri nets and adding labels to transitions to define the computation done by them. Then we associate NRC values with the tokens to represent the manipulated data. As it is usual with workflows that are described by Petri nets we mandate one special input place and one special output place. If there is external communication, this is modeled by transitions that correspond to calls to external functions. We use edge labeling to define how values of the consumed tokens map onto the parameters of operations represented by transitions. To express conditional behavior we propose edge annotations indicating conditions that the value associated with a token must satisfy, so it can be transferred through the annotated edge. We also introduce a special unnest/nest annotation, to enable explicit iteration over values of a collection.

A dataflow — a COSW in DFL — will be defined by an acyclic workflow net, transition labeling, edge labeling, and edge annotation. The underlying Petri net will be called a *dataflow net*.

Definition 3.3.1 (dataflow net)**.** A $DFN = \langle P, T, E, source, sink \rangle$ is a dataflow net if and only if:

(i) $\langle P, T, E \rangle$ is a workflow net and is acyclic,

(ii) $source \in P$ is the source place,

(iii) $sink \in P$ is the sink place.

The restriction to acyclic nets is introduced to keep the presentation of the main ideas simple. The formalism can be easily extended in such a way that

cycles are allowed. Usually they are used to express iteration over all elements of a collection, but for this type of iteration we will introduce alternatives in the form of constructs for unnesting and nesting values. Obviously this does not cover all types of iteration, but we conjecture that it is sufficient for the purpose of COSWs. In addition, an advantage of the restriction is that the termination is always guaranteed, but it has to be noted that termination does not ensure correct termination, which is defined as termination with only one token left which is in the sink and contains the output value.

3.3.1 The type system

Dataflows are strongly typed, which here means that each transition consumes and produces tokens with values of a well determined type. The type of the value of a token is called the *token type*. We will identify a type and the set of objects of that type. The type system is similar to that of NRC. We assume a finite but user-extensible set of basic types which might for example be given by:

$$b ::= \mathit{boolean} \mid \mathit{integer} \mid \mathit{string} \mid \mathit{XML}$$

where the type *boolean* contains the boolean values *true* and *false*, *integer* contains all integer numbers, *string* contains all strings and *XML* contains all well-formed XML documents. Although this set can be arbitrarily chosen we will require that it at least contains the *boolean* type. Assuming that the non-terminal l denotes the set of field labels, from these basic types we can build complex types as defined by:

$$\tau ::= b \mid \langle l : \tau, \ldots, l : \tau \rangle \mid \{\tau\}$$

The type $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$, where l_i are distinct labels, is the type of all records having exactly fields $l_1, \ldots, l_n$ of types $\tau_1, \ldots, \tau_n$ respectively (records with no fields are also included). Finally, $\{\tau\}$ is the type of all finite sets of elements of type τ. For later use we define CT to be the set of all complex types and CV the set of all possible complex values.

NRC can be also defined on other collection types such as lists or bags. Moreover they are included in existing COSW systems, for example Taverna supports lists. However, after a careful analysis of various use cases in bioinformatics and examples distributed with existing COSW systems we have concluded that sets are sufficient.

3.3.2 Edge naming function

Dataflows are not only models used to reason about data-processing experiments but are meant to be executed and produce computation results. In particular, when a transition has several input edges, we need a way to distinguish those, so as to know how the tokens map onto the operation arguments. This is solved by edge labeling. Only edges leading from places to transitions are labeled. This labeling is determined by an edge naming function $EN : \circ T \to EL$ (note that $\circ T = P\circ$), where EL is some countably infinite set of edge label names, e.g., all strings over a certain non-empty alphabet. The function EN is injective when restricted to incoming edges of a certain transition, i.e., there cannot be two distinct incoming edges with the same edge label for the same transition.

3.3.3 Transition naming function

To specify the desired operations and functions we also label the transitions. The transition labeling is defined by a transition naming function $TN : T \to TL$, where TL is a set of transition labels. Each transition label determines the number and possible labeling of input edges as well as the types of tokens that the transition consumes and produces when it fires. For this purpose the input typing and output typing functions are used: $IT : TL \to CT$ maps each transition label to the input type which must be a record type, and $OT : TL \to CT$ maps each transition label to the output type. Note that these two functions are at the global level in the sense that they are the same for every dataflow and therefore not part of the dataflow itself. This is similar to the signatures of system functions which are not part of a specific program. For detailed specification of transition labels see Section 3.4.

3.3.4 Edge annotation function

To introduce conditional behavior we annotate edges with conditions. If an edge is annotated with a condition, then it can only transport tokens that satisfy the condition. Conditions are visualized on diagrams in UML [44] fashion, i.e., in square brackets. Only edges leading from places to transitions are annotated with conditions. There are four possible condition annotations: "=true", "=false", "$=\emptyset$", "$\neq\emptyset$". Their meaning is self-explanatory. For detailed specification see Section 3.5.

There is another annotation "$*$" used to indicate a special unnest/nest branch. On diagrams it is visualized by addition of the symbol "$*$" in the middle of the edge. This annotation can occur on edges leading from transitions to places as well as on edges from places to transitions. When an edge leading from a transition to a place is annotated in such manner, it means that a set value produced by this transition is unnested. That is, instead of inserting a token with a set value into the destination place, a set of tokens representing each element in the set value gets inserted. Such edges will be called *unnest edges*. When an edge leading from a place to a transition is annotated in such manner, it means that in order to fire the destination transition a set of tokens that originated from unnesting of some set value will be used. That is, a set of tokens that originated from unnesting of some set value will be consumed and a set of their values will be an input data for the destination transition. Such edges will be called *nest edges*. The precise semantics and explanation of the mechanism that is used to make sure that all the tokens that originated from unnesting of some set value are already there is described in Section 3.5.

The annotations are defined by an edge annotation function:

$$EA : (\circ T \cup \circ P) \to \{\text{“=true”}, \text{“=false”}, \text{“}=\emptyset\text{”}, \text{“}\neq\emptyset\text{”}, \text{“}*\text{”}, \varepsilon\},$$

where ε indicates the absence of an annotation and for all $e \in \circ T$ it holds that $EA(e) \in \{\text{“=true”}, \text{“=false”}, \text{“}=\emptyset\text{”}, \text{“}\neq\emptyset\text{”}, \text{“}*\text{”}, \varepsilon\}$ while for all $e \in \circ P$ it holds that $EA(e) \in \{\text{“}*\text{”}, \varepsilon\}$. From now on we will use the following notation to stress the last two properties:

$$EA : (\circ T \to \{\text{“=true”}, \text{“=false”}, \text{“}=\emptyset\text{”}, \text{“}\neq\emptyset\text{”}, \text{“}*\text{”}, \varepsilon\}) \uplus (\circ P \to \{\text{“}*\text{”}, \varepsilon\}),$$

3.3.5 Place type function

With each place in a dataflow net we associate a specific type that restricts the allowed values for tokens in that place. This is represented by a place type function $PT : P \to CT$.

3.3.6 Dataflow

The dataflow net with edge naming, transition naming, edge annotation and place typing functions specifies a dataflow.

Definition 3.3.2 (dataflow)**.** A dataflow is a five-tuple $\langle DFN, EN, TN, EA, PT\rangle$ where:

- $DFN = \langle P, T, E, source, sink\rangle$ is a dataflow net,
- $EN : {\circ}T \to EL$ is an edge naming function such that for each transition t the partial function $EN|_{{\circ}t}$ is injective,
- $TN : T \to TL$ is a transition naming function,
- $EA : ({\circ}T \to \{\text{“=true”}, \text{“=false”}, \text{“}=\emptyset\text{”}, \text{“}\neq\emptyset\text{”}, \text{“}*\text{”}, \varepsilon\}) \uplus ({\circ}P \to \{\text{“}*\text{”}, \varepsilon\})$ is an edge annotation function,
- $PT : P \to CT$ is a place type function.

In order to ensure that the different labellings and annotations in a dataflow are consistent, we introduce the notion of *legality*. Informally, a dataflow is legal, if for each transition t: (1) the input edge labels and the types of their corresponding places, with the nest edges taken into account, define the input type of t; (2) if any of the input edges of t are annotated with conditions, then the annotations are consistent with the types of the associated input places; (3) if an output edge of t is not an unnest edge, then the type of the connected place is equal to the output type of t, but if an output edge of t is an unnest edge, then the output type of t is a set type and the type of the connected place is equal to the element type of this set type.

Definition 3.3.3 (legal)**.** A dataflow $\langle DFN, EN, TN, EA, PT\rangle$ is legal if and only if each transition $t \in T$ satisfies the following:

1. if $\{\langle p_1, t\rangle, \ldots, \langle p_n, t\rangle\} = {\circ}t$ and for $1 \leq i \leq n$ we have
   $l_i = EN(\langle p_i, t\rangle)$ and $\tau_i = \begin{cases} PT(p_i) & \text{if } EA(\langle p_i, t\rangle) \neq \text{“}*\text{”} \\ \{PT(p_i)\} & \text{if } EA(\langle p_i, t\rangle) = \text{“}*\text{”} \end{cases}$
   then $IT(TN(t)) = \langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle$,
2. for each $\langle p, t\rangle \in {\circ}t$:
   - if $EA(\langle p, t\rangle) \in \{\text{“=true”}, \text{“=false”}\}$, then $PT(p) = \mathit{boolean}$, and
   - if $EA(\langle p, t\rangle) \in \{\text{“}=\emptyset\text{”}, \text{“}\neq\emptyset\text{”}\}$, then $PT(p)$ is a set type,
3. for each $\langle t, p\rangle \in t{\circ}$:

- if $EA(\langle t, p\rangle) \neq$ "$*$", then $OT(TN(t)) = PT(p)$, and
- if $EA(\langle t, p\rangle) =$ "$*$", then $OT(TN(t)) = \{PT(p)\}$.

Henceforth, dataflows will always be assumed to be legal. Legality is an easy syntactic check.

An example dataflow representing an **if** $u = v$ **then** $f(x)$ **else** $g(x)$ expression is shown in Fig. 3.2. Although the transition labels and a precise execution semantics are defined in the next two sections, the example is self-explanatory. First, three copies of the input record of type $\langle u : b, v : b, x : \tau\rangle$ are made. Then, each copy is projected onto another field, basic values u and v are compared, and a choice between upper or lower dataflow branch is made on the basis of the boolean comparison result. The boolean value is disposed in a projection and depending on the branch that was chosen either $f(x)$ or $g(x)$ is computed.

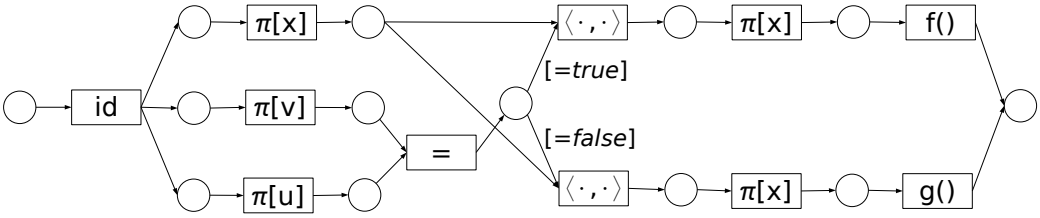

Figure 3.2: If-then-else example

3.4 Transition labels

Since it is impossible to gather all scientific analysis tools that one may want to use and data repositories that one may want to query, DFL defines only a *core label subset* that is sufficient to express typical operations on the values from our type system. Similarly to NRC, DFL can be extended with new *extension transition labels*. Such extension labels will usually represent computations done by external services. Examples from the domain of bioinformatics include: sequence similarity searches with BLAST [3], queries to the Swiss-Prot [8] protein knowledgebase, or local enactments of the tools from the EMBOSS [52] package.

3.4.1 Core transition labels

The core transition labels are based on the NRC operator set and are shown in Table 3.1. A transition label is defined as a combination of the basic symbol, from the first column, and a list of parameters which consists of types and edge labels, from the second column. The values of the input type function IT and the output type function OT are given by the last two columns. For example, a concrete instance of the record constructor label, i.e., a constructor label with concrete parameter values, would be $tl' = \langle \cdot, \cdot \rangle_{a,bool,b,int}$ where the parameters are indicated in subscript and for which the functions IT and OT are defined such that $IT(tl') = OT(tl') = \langle a : bool, b : int \rangle$. This means that tl' has to have two input edges labeled a and b, first of which connects to a place of type *bool* and second to a place of type *int*. Another example would be $tl'' = \pi[b]_{a,b,bool,c,int}$, where $IT(tl'') = \langle a : \langle b : bool, c : int \rangle \rangle$ and $OT(tl'') = bool$.

Table 3.1: Core transition labels

Sym.	**Param.**	**Oper. name**	**Input type**	**Output type**
$\emptyset$	l, τ_1, τ_2	empty-set constr.	$\langle l : \tau_1 \rangle$	$\{\tau_2\}$
$\{\cdot\}$	l, τ	singleton-set constr.	$\langle l : \tau \rangle$	$\{\tau\}$
$\cup$	l_1, l_2, τ	set union	$\langle l_1 : \{\tau\}, l_2 : \{\tau\} \rangle$	$\{\tau\}$
φ	l, τ	flatten	$\langle l : \{\{\tau\}\} \rangle$	$\{\tau\}$
$\times$	l_1, τ_1, l_2, τ_2	Cartesian product	$\langle l_1 : \{\tau_1\}, l_2 : \{\tau_2\} \rangle$	$\{\langle l_1 : \tau_1, l_2 : \tau_2 \rangle\}$
$=$	l_1, l_2, b	atomic-value equal.	$\langle l_1 : b, l_2 : b \rangle$	*boolean*
$\langle\rangle$	l, τ	empty record constr.	$\langle l : \tau \rangle$	$\langle\rangle$
$\langle \cdot, \cdot \rangle$	$l_1, \tau_1, \ldots, l_n, \tau_n$	record constr.	$\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$	$\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$
$\pi[l_i]$	$l, l_1, \tau_1, \ldots, l_n, \tau_n$	field projection	$\langle l : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \rangle$	τ_i
id	l, τ	identity	$\langle l : \tau \rangle$	τ

Moreover, for every transition label tl, there exists an associated function $\Phi_{tl} : IT(tl) \rightarrow OT(tl)$ which represents a computation that is performed when the transition fires. For the core transition label subset all functions are deterministic and correspond to those given in NRC definition [9].

To keep this presentation simple we will omit edge names and label parameters on diagrams, if it does not introduce ambiguity.

3.4.2 Extension transition labels

Next to the set of core transition labels, the set of transition labels TL also contains user-defined transition labels. As for core transition labels the functions IT and OT must be defined for each of them, as well as an associated function $\Phi_{tl} : IT(tl) \rightarrow OT(tl)$ which can represent a possibly non-deterministic computation that is performed when the transition fires.

To give a concrete example, a $getSWPrByAC$ extension transition label may be defined by a bioinformatician, such that $IT(getSWPrByAC) = \langle ac : string \rangle$ and $OT(getSWPrByAC) = XML$. The $\Phi_{getSWPrByAC}$ function would represent a call to a Swiss-Prot knowledgebase and return an XML formatted entry for a given primary accession number.

3.5 Transition system semantics

The semantics of a dataflow $\langle DFN, EN, TN, EA, PT \rangle$ is defined as a transition system (see Section 3.5.2). Each place contains zero or more *tokens*, which represent data values. Formally a token is a pair $k = \langle v, h \rangle$, where $v \in CV$ is the transported value and h the *unnesting history* of this value. This unnesting history is defined in Section 3.5.1. The set of all possible tokens is then $K = CV \times H$. By the type of a token we mean the type of its value, i.e., $\langle v, h \rangle : \tau$ if and only if $v : \tau$.

The state of a dataflow, also called *marking*, is the distribution $M : (P \times K) \rightarrow (\mathbb{N} \cup \{0\})$ of tokens over places, where $M(p, k) = n$ means that place p contains n copies of the token k[1]. Distributions are legal as markings only if the token types match the types of places they are in, i.e., for all places $p \in P$ and tokens $k \in K$ such that $M(p, k) > 0$ we must have $k : PT(p)$.

Transitions are the active components in a dataflow. They can change the state by *firing*, i.e., consuming tokens from their input places and producing tokens in their output places. In distinction to classical workflow nets, transitions may produce/consume an arbitrary number of tokens in/from a place. This is the case when an edge connecting such a place with the transition is annotated with "$*$", i.e., is an unnest/nest edge. A transition that can fire in

[1] Although formally, because K is infinite, the definition allows for infinite markings, we will only consider markings M where the *support* $\{\langle p, k \rangle \in (P \times K) \mid M(p, k) \neq 0\}$ is finite. It is easy to see, that during a dataflow execution the finiteness of support does not change.

a given state will be called *enabled*. The firing of a transition t represents a computation step determined by the function $\Phi_{TN(t)}$ associated with its transition label. Values carried by tokens consumed from input places together with labels of the corresponding input edges determine the input record value for the function with respect to the definitions in Table 3.1.

3.5.1 Token unnesting history

Every time a transition with an unnest edge fires, a set of tokens is produced. Each token corresponds to an element of the set value that was produced as a result of a computation carried out by that transition. The history of each of the tokens is extended with a pair that contains the unnested set and the element of that set to which the given token corresponds. The full history is taken into account when it is being determined whether a transition with nest edge can fire, that is if tokens representing all of the elements of the set that is being nested are already there to be consumed. If this is the case, then a set of tokens will be consumed and the set of their values will be used to compute the result.

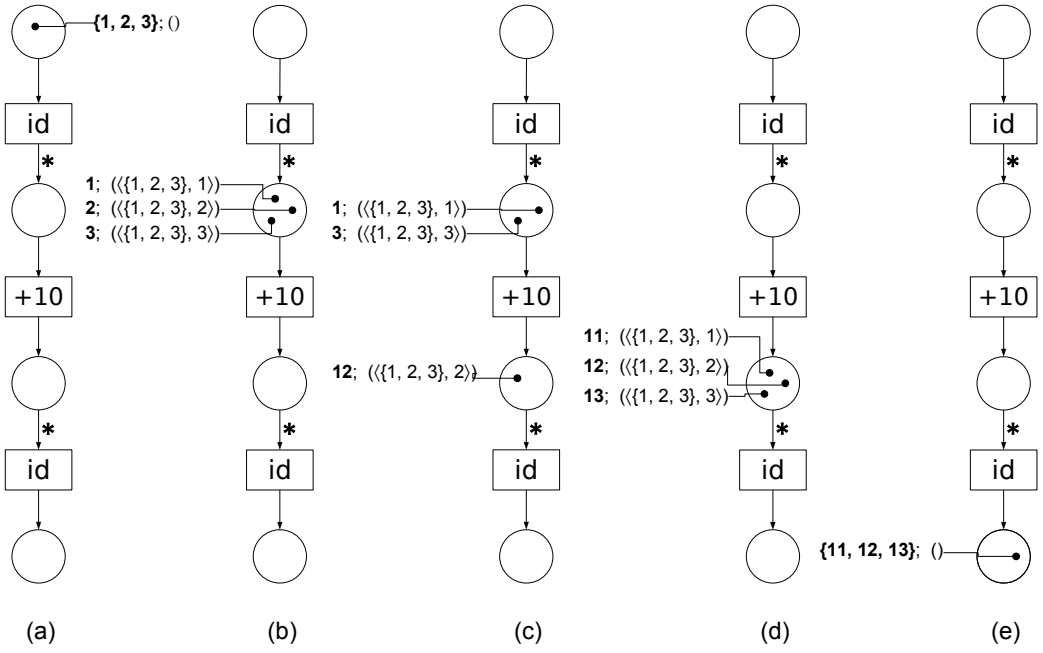

Figure 3.3: An illustration of the unnest/nest edges and the unnesting history

This is illustrated in Fig. 3.3 where in (a) in the top place we see a single token with value $\{1, 2, 3\}$ and an empty history. When the upper *id* transition fires, a token for each element of the output value $\{1, 2, 3\}$ is produced as shown in (b). The history is extended at the end with a pair that contains, first, the set that was unnested and, second, the element for which this particular token was produced. As is shown in (b), (c) and (d) transitions without any unnest or nest edge will produce tokens with histories identical to that of the consumed input tokens. Once all the tokens that belong to the same unnesting group have arrived in the input place of the bottom *id* transition, as is shown in (d), it can fire and combine them into a single set-valued token as is shown in (e). A transition can verify if all the tokens that belong to the same unnesting group have arrived by looking at their histories. Note that where the firing of a transition with an unnest edge adds a pair to the history, firing a transition with a nest edge removes a pair from the history.

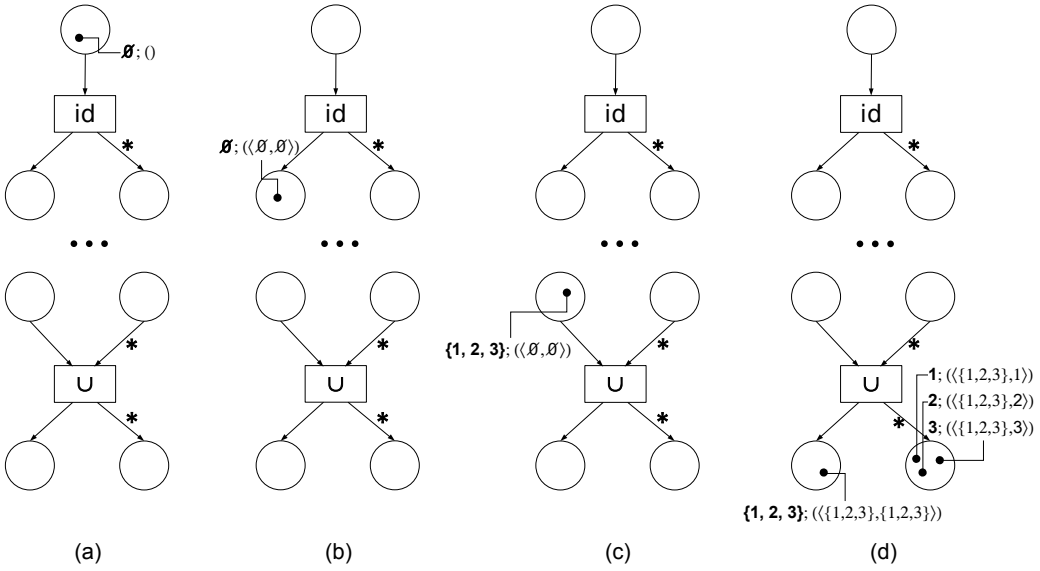

Figure 3.4: An illustration of the unnesting history and iteration over empty sets

The second example (see Fig. 3.4) presents what happens when one transition has unnest/nest edges as well as normal edges. The initial state is presented in (a). As shown in (b), after firing transition *id*, the token representing an empty set has been consumed. Since *id* has an unnest edge,

the result of its computation — an empty set — has been unnested and zero tokens have been inserted into the right output place. Yet, the left output place is connected by a normal edge and a token has been produced there. Because unnesting has been performed on the "$*$" annotated edges, its history has been extended with a pair consisting of twice the unnested set. After some additional processing this token transports a set of three numbers $\{1, 2, 3\}$ as can be observed in (c). Now the set union transition can fire. Although one of its input places is empty, it is enabled because it is connected by a nest edge and the examination of the history of the token from the other input place that was connected by a normal edge shows that tokens representing elements of an empty set are to be expected there (so no tokens need to be consumed). When the set union transition fires, a set of $\{1, 2, 3\}$ will be produced as a result of the union of $\{1, 2, 3\}$ with an empty set. As is shown in (d) another unnest can be performed and this time tokens are inserted to both output places.

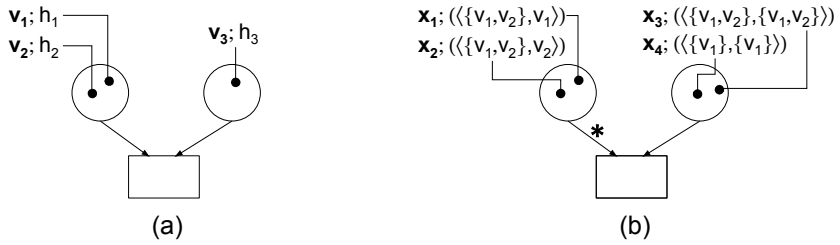

Figure 3.5: An illustration of how history affects transitions with many input places

In the case of transitions with many input edges tokens consumed from distinct input places must either have the same history or must represent the same set. This way the history of the tokens produced by such a transition can be unambiguously determined, tokens representing elements of different sets do not interfere with each other in the body of the iteration and at the same time the order of execution is free of any unnecessary restrictions. This is illustrated on the third example (see Fig. 3.5). The transition in (a) can fire only if $h_1 = h_3$ or $h_2 = h_3$. Otherwise it is not enabled even though some tokens are in both of its input places. The transition in (b) can fire consuming tokens with values x_1 and x_2 from the left input place and x_3 from the right input place since they represent the same sets. A token with value x_4 cannot be consumed in this state, because there is no token representing

the element of set $\{v_1\}$ in the left input place.

Since sets can be unnested and nested several times, the history is a *sequence* of pairs, where each pair contains the unnesting information of one unnesting step. Therefore we formally define the *set of all histories* H as the set of all sequences of pairs $\langle s, x\rangle$, where $s \in CV$ is a set and $x \in s$ or $x = s$. To manipulate histories we will use the following notation for extending a sequence with an element $(a_1, a_2, \ldots, a_n) \oplus a_{n+1} := (a_1, a_2, \ldots, a_n, a_{n+1})$.

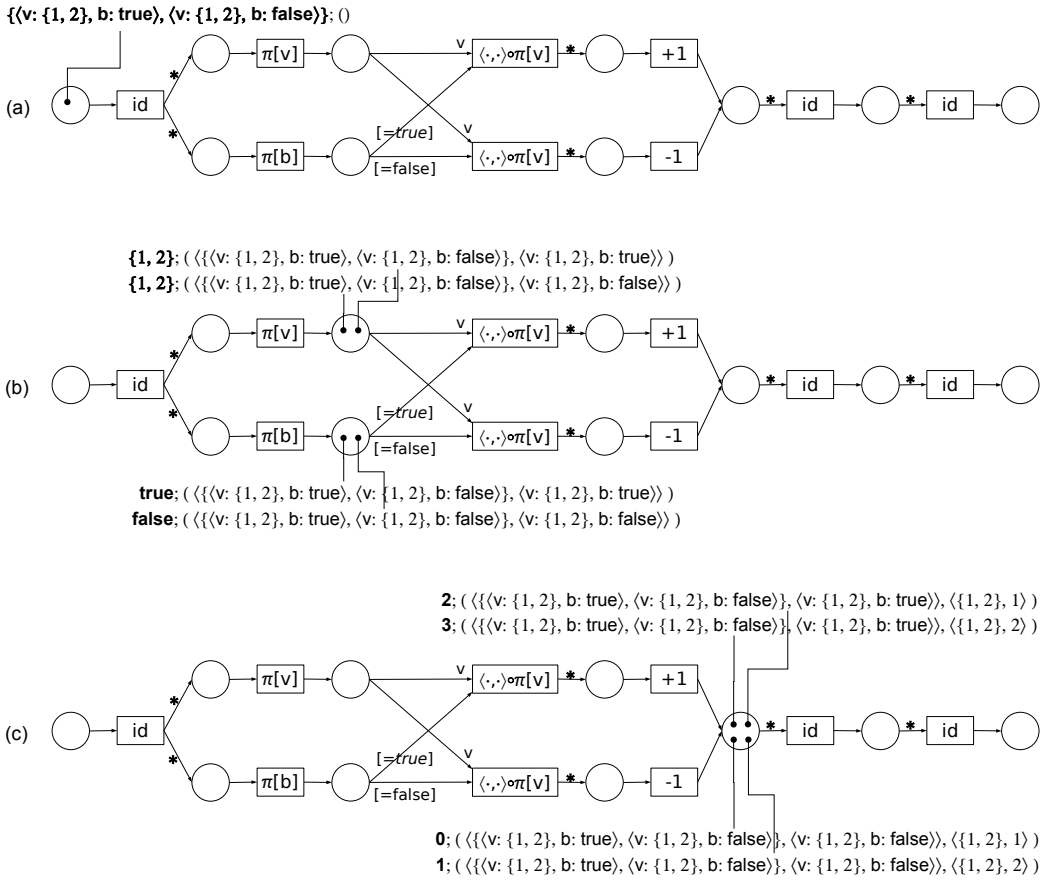

Figure 3.6: An illustration of a nested iteration

The fourth and final example demonstrates why the whole history and not only its last element is taken into account while nesting. The dataflow in Fig. 3.6 unnests the input set of type $\{\langle v : \{integer\}, b : boolean\rangle\}$ and processes each of its pair values based on the boolean element. For pairs

with a *true* value, every element of the associated set of integers is increased by one, while for pairs with a false value, the elements are decreased by one. In (a) the initial state with the input value is presented. In (b) the input value has been already unnested and, similarly as with the If-then-else example from Section 3.3.6, the paired elements have been separated to make the trueness based test. The $\langle \cdot, \cdot \rangle \circ \pi[v]$ transitions are used to dispose of the boolean value by creating a pair and projecting the boolean value out. Although in this example it is not important, since both pairs contained the same set $\{1, 2\}$, the transitions labeled $\langle \cdot, \cdot \rangle \circ \pi[v]$ would not consume values with different histories thus retaining the original pairing. In (c) the integer sets have been unnested and their values have been increased in the upper branch and decreased in the lower branch. The processed values are gathered in one place and are ready to be nested back. Observe that inspecting the last element of the history during nesting is not enough and the whole history has to be taken into account to prevent intermixing of the values processed by the lower and the upper branch.

It should be noted that this provides the dataflow authors with a simple way to achieve *multiple instantiation* [25], which in the process modeling domain is known as the ability to execute a task multiple times, possibly simultaneously. Conveniently our approach does not enforce iterating over elements of a set in any particular order and the transition semantics is local. At the same time it is always possible to determine if a given transition can fire and even in the case of a nested iteration over nested sets, tokens representing elements of different sets will not become intermixed.

3.5.2 Semantics of transitions

We define the semantics as a transition system, where the states are the distributions of tokens over places and state changes are caused by firing enabled transitions. A transition is *enabled* in a given state, if from each of its input places it can consume tokens with matching histories — an arbitrary number from places connected by nest edges or one if it is not the case. Those tokens/sets of tokens represent values that will become arguments for the function represented by the enabled transition. The choice of such tokens and the function arguments determined by it are called an *enabling configuration*.

The following shortcut will be used, since tokens can only flow along a

condition-annotated edge, if the value of the token satisfies the condition:

$$\begin{aligned}\langle v, h\rangle \curvearrowright e \quad \stackrel{\text{def}}{=} \quad & (EA(e) = \varepsilon) \vee (EA(e) = \text{“}*\text{”}) \vee \\ & (EA(e) = \text{“=true”} \wedge v = \text{true}) \vee \\ & (EA(e) = \text{“=false”} \wedge v = \text{false}) \vee \\ & (EA(e) = \text{“}=\emptyset\text{”} \wedge v = \emptyset) \vee \\ & (EA(e) = \text{“}\neq\emptyset\text{”} \wedge v \neq \emptyset)\end{aligned}$$

Definition 3.5.1 (enabling configuration)**.** Given a transition t in marking M, an enabling configuration is a function $EC : {\bullet t} \to 2^K$ such that:

(i) for all places $p \in {\bullet t}$ and for all tokens $k \in EC(p)$ it holds that $M(p, k) \geq 1$ and $k \curvearrowright \langle p, t\rangle$,

(ii) at least one token is in the range of EC, i.e., $\bigcup_{p \in {\bullet t}} EC(p) \neq \emptyset$, and

(iii) there is a history h such that:

- if t has at least one nest edge, then there exists a set $S = \{x_1, \ldots, x_m\} \in CV$ such that for all places $p \in {\bullet t}$ it holds that
$$EC(p) = \begin{cases} \{\langle v_{p,1}, h \oplus \langle S, x_1\rangle\rangle, \ldots, \langle v_{p,m}, h \oplus \langle S, x_m\rangle\rangle\} & \text{if } EA(\langle p, t\rangle) = \text{“}*\text{”} \\ \{\langle v_p, h \oplus \langle S, S\rangle\rangle\} & \text{if } EA(\langle p, t\rangle) \neq \text{“}*\text{”} \end{cases}$$
for some complex values $v_{p,1}, \ldots, v_{p,m}$ and v_p,
- if t has no nest edge, then for all places $p \in {\bullet t}$ it holds that $EC(p) = \{\langle v_p, h\rangle\}$ for some complex value v_p.

Note that since the range of the enabling configuration contains at least one token, it holds that if such an EC exists, then h is uniquely determined, so we denote it as h_{EC}.

Moreover, given such an EC we define the *enabling configuration value function* $ECV_{EC} : {\bullet t} \to CV$, which with a place p associates the value represented by the tokens pointed to by $EC(p)$, i.e., for all places $p \in {\bullet t}$ it holds that

$$ECV_{EC}(p) = \begin{cases} \{v_{p,1}, \ldots, v_{p,m}\} & \text{if } EA(\langle p, t\rangle) = \text{“}*\text{”} \\ v_p & \text{if } EA(\langle p, t\rangle) \neq \text{“}*\text{”}. \end{cases}$$

A transition for which an enabling configuration exists can fire and it is called *enabled*. In a given state many enabling configurations can exist for

one transition. For example, if t has two input places connected by normal edges, one of its input place contains two tokens, the other contains three tokens and all the tokens have the same history, then there exist six enabling configurations for t in this state.

Definition 3.5.2 (enabled transition)**.** Transition t is enabled in a given marking M if and only if there exists an enabling configuration for t in M.

When a transition fires, it consumes tokens according to some enabling configuration EC and the transition's associated function is being computed with the arguments pointed to by ECV_{EC}.

State transition (firing a transition)

For each $t \in T$ it holds that $M_1 \xrightarrow{t} M_2$ if and only if there exists an enabling configuration EC for t in marking M_1 such that

1. for all places $p \in \bullet t$ it holds that:
   (a) $M_2(p,k) = M_1(p,k) - 1$ if $k \in EC(p)$, and
   (b) $M_2(p,k) = M_1(p,k)$ if $k \notin EC(p)$
2. if t has no unnest edges, then for all places $p \in t\bullet$ it holds that, if v_{res} is the result of $\Phi_{TN(t)}(\langle l_1 : v_1, \ldots, l_n : v_n\rangle)$, in case when $\Phi_{TN(t)}$ is a deterministic function, or one of its possible results, when it is non-deterministic, where $\{\langle l_1, v_1\rangle, \ldots, \langle l_n, v_n\rangle\} = \{\langle EN(\langle p', t\rangle), ECV_{EC}(p')\rangle \mid p' \in \bullet t\}$ then:
   (a) $M_2(p, \langle v_{res}, h_{EC}\rangle) = M_1(p, \langle v_{res}, h_{EC}\rangle) + 1$, and
   (b) $M_2(p, \langle v', h'\rangle) = M_1(p, \langle v', h'\rangle)$ if $\langle v', h'\rangle \neq \langle v_{res}, h_{EC}\rangle$
3. if t has at least one unnest edge, then for all places $p \in t\bullet$ it holds that, if v_{res} is the result of $\Phi_{TN(t)}(\langle l_1 : v_1, \ldots, l_n : v_n\rangle)$, in case when $\Phi_{TN(t)}$ is a deterministic function, or one of its possible results, when it is non-deterministic, where $\{\langle l_1, v_1\rangle, \ldots, \langle l_n, v_n\rangle\} = \{\langle EN(\langle p', t\rangle), ECV_{EC}(p')\rangle \mid p' \in \bullet t\}$ then:
   (a) $M_2(p, \langle v_{res}, h_{EC} \oplus \langle v_{res}, v_{res}\rangle\rangle) = M_1(p, \langle v_{res}, h_{EC} \oplus \langle v_{res}, v_{res}\rangle) + 1$ if $EA(\langle t, p\rangle) \neq$ "$*$", and

(b) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $EA(\langle t, p \rangle) \neq$ "$*$" and $\langle v', h' \rangle \neq \langle v_{res}, h_{EC} \oplus \langle v_{res}, v_{res} \rangle \rangle$

(c) $M_2(p, \langle v, h_{EC} \oplus \langle v_{res}, v \rangle \rangle) = M_1(p, \langle v, h_{EC} \oplus \langle v_{res}, v \rangle) + 1$ if $EA(\langle t, p \rangle) =$ "$*$" and $v \in v_{res}$, and

(d) $M_2(p, \langle v', h' \rangle) = M_1(p, \langle v', h' \rangle)$ if $EA(\langle t, p \rangle) =$ "$*$" and $\langle v', h' \rangle \neq \langle v, h_{EC} \oplus \langle v_{res}, v \rangle \rangle$ for all $v \in v_{res}$

4. for all places $p \notin {\bullet t} \cup t{\bullet}$ it holds that $M_2(p, k) = M_1(p, k)$ for all tokens $k \in K$

It should be noted that for a given state M_1, a transition t and two not equal states M_2 and M_3 it can hold that $M_1 \xrightarrow{t} M_2$ and $M_1 \xrightarrow{t} M_3$. This is because in M_1 there can be more than one enabling configuration for t. It can also be the case that the function represented by t is not a deterministic one and transitions to M_2 and M_3 are possible for the same enabling configuration, because two different output values can be produced.

We adopt the following Petri net notations:

- $M_1 \rightarrow M_2$: there is a transition t such that $M_1 \xrightarrow{t} M_2$
- $M_1 \xrightarrow{\theta} M_n$: the firing sequence $\theta = t_1 t_2 \dots t_{n-1}$ leads from state M_1 to state M_n, i.e., $\exists_{M_2, M_3, \dots, M_{n-1}} M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \xrightarrow{t_3} \dots \xrightarrow{t_{n-1}} M_n$
- $M_1 \xrightarrow{*} M_n$: $M_1 = M_n$ or there exists a firing sequence $\theta = t_1 t_2 \dots t_{n-1}$ such that $M_1 \xrightarrow{\theta} M_n$

A state M_n is called *reachable* from M_1 if and only if $M_1 \xrightarrow{*} M_n$.

Although the semantics of a dataflow is presented as a transition system, as in classical Petri nets, two or more enabled transitions may fire concurrently, if there are enough input tokens for both of them.

3.6 A bioinformatics example

In this section we[2] present a dataflow example based on a part of a real bioinformatics example [19]. The dataflow is shown in Fig. 3.7. Its goal is to find differences in peptide content of two samples of cerebrospinal fluid (a peptide

[2]The example was proposed by Natalia Kwasnikowska.

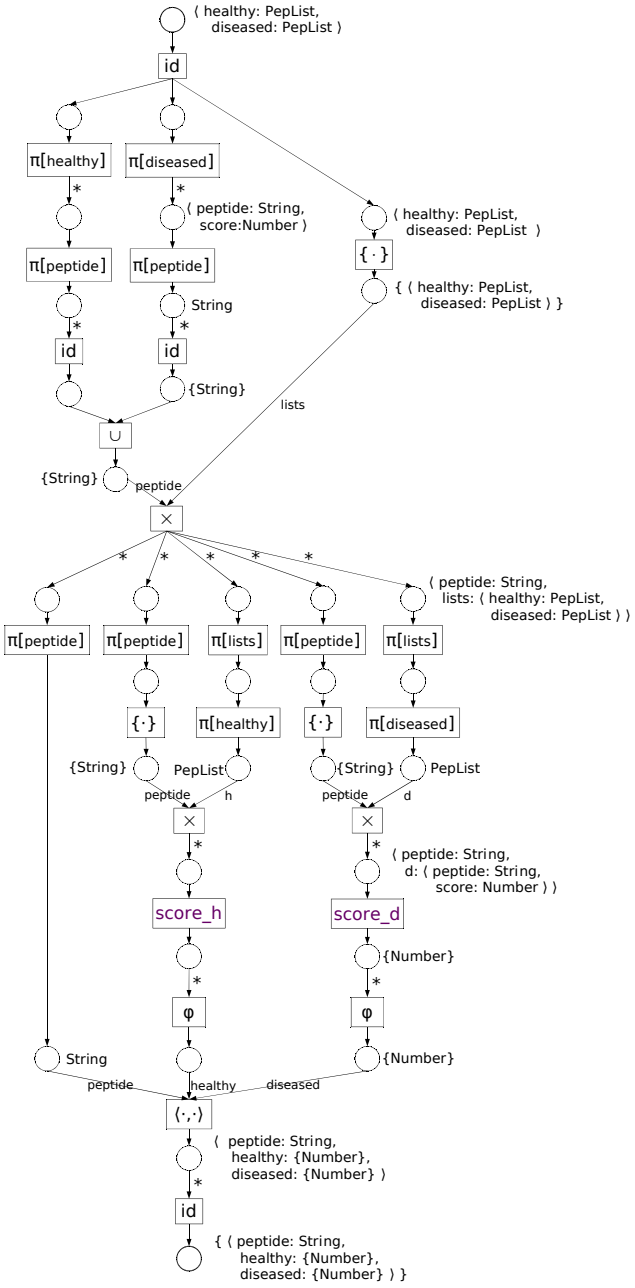

Figure 3.7: Finding differences in peptide content of two samples

is an amino acid polymer). One sample belongs to a diseased person and the other to a healthy one. A mass spectrometry wet-lab experiment has provided data about observed polymers in each sample. A peptide-identification algorithm was invoked to identify the sequences of those polymers, providing an amino-acid sequence and a confidence score for each identified polymer.

The COSW starts with a record value containing two sets of data from the identification algorithm, one obtained from the "healthy" sample and the other from the "diseased" sample: complex input type $\langle$ `healthy` : *PepList*, `diseased` : *PepList* $\rangle$ where *PepList* is the complex type $\{\langle$ `peptide` : **String**, `score` : **Number** $\rangle\}$. Each data set contains records consisting of an identified peptide, represented by the basic type **String**, and the associated confidence score, represented by the basic type **Number**. The COSW transforms this input into a set of records containing the identified peptide, a singleton containing the confidence score from the "healthy" data set or an empty set if the identified peptide was absent in the "healthy" data set, and similarly, the confidence score from the "diseased" data set. The complex output type is the following: $\{\langle$ `peptide` : **String**, `healthy` : $\{$ **Number** $\}$, `diseased` : $\{$ **Number** $\}\rangle\}$.

The global structure of the dataflow can be described as follows. In the first part up to and including the first transition labeled $\times$ it computes the Cartesian product of two sets. The first set is computed in the left branch, which consists again of two sub-branches, and is the union of all mentioned peptides in the initial record. The second set is computed in the right branch and is the singleton set containing the initial record. In the second part of the dataflow, between the first Cartesian product and the final place, the dataflow iterates over the result of the first part and processes the records in the Cartesian product in parallel in three branches, where the rightmost two branches themselves consist of two sub-branches, and combines their results into a single record with a record constructor. The first branch simply projects the record on the peptide label. The second and third branch compute the scores of this peptide in the "healthy" peptide list and the "diseased" peptide list, respectively. They do so by computing the Cartesian product of the peptide and the relevant peptide list, iterating over the result and applying to each record the function score_h (or score_d) which compares the first peptide with the peptide in the nested record and if they are equal returns a singleton set with the score or an empty set otherwise. Note that the transitions labeled score_h and score_d could have

been decomposed further and replaced with dataflows, but are represented here by single transitions for brevity. Finally the dataflow collects all the records consisting of the peptide and its scores in the "healthy" and the "diseased" peptide list, into a single set.

3.7 Hierarchical collection-oriented scientific workflows

Our extension of workflow nets allows the reuse of various technical and theoretical results that are known about them. This is what we intend to demonstrate here by discussing a way of constructing dataflows that guarantees that they always satisfy certain correctness criteria. A well-known technique for this is the use of refinement rules that allow the step-wise generation of Petri nets by replacing a transition or place with a slightly bigger net. Such refinement rules were studied by Berthelot in [7] and Murata in [41] as reduction rules that preserve liveness and boundedness properties of Petri nets. They are used by van der Aalst in [66], by Reijers in [50] and by Chrząstowski-Wachtel et al. in [13] to generate workflow nets. We show that the same principles can be applied to our extended notion of workflow net, and can be adapted to deal with the new problem of data-dependent control flow.

DFL is developed to model data-centric workflows and in particular scientific data-processing experiments. The data to be processed should be placed in the dataflow's source and after the processing, the result should appear in its sink. A special notation is introduced to distinguish between two state families.

Definition 3.7.1 (input state)**.** Given dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ with $DFN = \langle P, T, E, source, sink \rangle$ and value $v : PT(source)$ we define *the input state* $input_v^D$ as a marking such that:

- $input_v^D(source, \langle v, () \rangle) = 1$, and
- for all places $p \in P$ and tokens $k \in K$ such that $\langle p, k \rangle \neq \langle source, \langle v, () \rangle \rangle$ it holds that $input_v^D(p, k) = 0$.

Definition 3.7.2 (output state)**.** Given dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ with $DFN = \langle P, T, E, source, sink \rangle$ and value $v : PT(sink)$ we define *the output state* $output_v^D$ as a marking such that:

- $output_v^D(sink, \langle v, () \rangle) = 1$, and
- for all places $p \in P$ and tokens $k \in K$ such that $\langle p, k \rangle \neq \langle sink, \langle v, () \rangle \rangle$ it holds that $output_v^D(p, k) = 0$.

Starting with one token in the source and executing the dataflow need not always produce a result in the form of a single token in the sink place. For some dataflows the computation may halt in a state in which none of the transitions is enabled, yet the sink is empty. For other dataflows the result token may be produced, but there still may be tokens left in other places. Furthermore, for some dataflows reaching a state in which there are no tokens at all is possible.

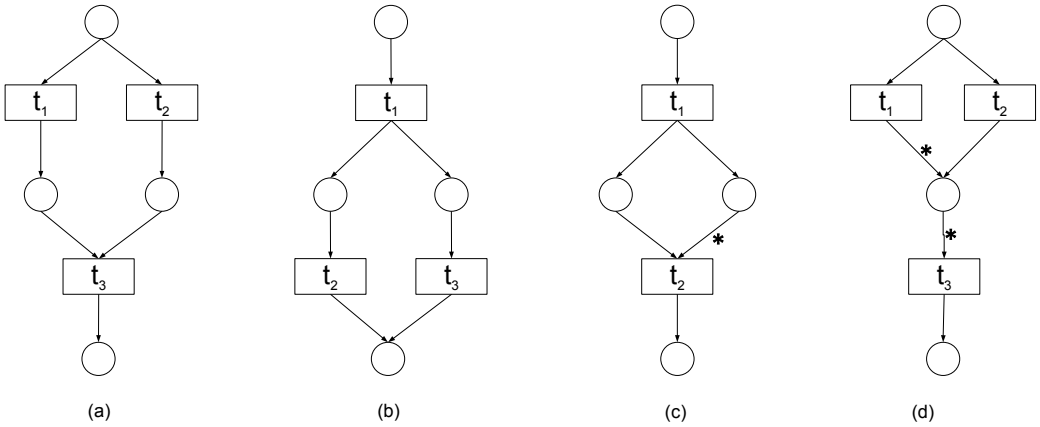

Figure 3.8: Examples of dataflows that may not finish properly

Examples of dataflows for which starting with one token does not always produce a result in the form of a single token in the sink place are shown in Fig. 3.8. For the dataflow (a) the token from the source can be consumed by a transition t_1 or t_2, but not by both of them at the same time. Transition t_3 will not become enabled then, because one of its input places will stay empty. The (b) case presents an opposite scenario. Transition t_1 produces two output tokens and after either t_2 or t_3 consumes one of them and produces a computation result, the second token is still there and another computation result can be produced. In the (c) case t_2 will never become enabled, since the tokens with history appropriate for nesting will never be produced by t_1. Similarly in case (d) if t_2 gets the source token, t_3 will not become enabled, because only t_1 can produce a token with the required history. But for (d)

it may even be not enough, when the t_1 consumes the source token. If the source token carried an empty set, then in the resulting state all places would be empty.

Similar problems were also studied in the context of procedures modeled by classical workflow nets. The procedures without such problems are called *sound* [1]. A workflow net is considered to be sound if an only if:

1. if after starting with a single token in the source place a token gets inserted into the sink, then there are no other tokens left,

2. the computation can be always completed, that is, if one starts with a single token in the source and regardless of how the computation proceeds at start, it is always possible to reach a state with the only token in the sink place, and

3. there are no transitions that cannot be fired if starting with a single token in the source place.

This classical notion of soundness can be directly applied to dataflows such as (a) and (b) in Fig. 3.8 where the control flow does not depend upon the data, but in dataflows such as (c) and (d) where the control flow may depend upon the values and the unnesting histories associated with a token the notion needs to be adapted. Here tokens carry values, so there are many possible input states from which a computation can be started — one for each possible value for the first token. It is natural to require that each transition becomes enabled after starting from some input state, but not from all.

Definition 3.7.3 (soundness)**.** A dataflow $D = \langle DFN, EN, TN, EA, PT\rangle$ with $DFN = \langle P, T, E, source, sink\rangle$ is sound if and only if:

(i) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \xrightarrow{*} M$, if for some value $v'' : PT(sink)$ and history $h'' \in H$ it holds that $M(sink, \langle v'', h''\rangle) > 0$, then $M = output^D_{v''}$,

(ii) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \xrightarrow{*} M$ there exists a value $v'' : PT(sink)$ such that $M \xrightarrow{*} output^D_{v''}$, and

(iii) for each transition $t \in T$ there exists a value $v' : PT(source)$ and two markings M and M' such that $input^D_{v'} \xrightarrow{*} M \xrightarrow{t} M'$.

Although it seems desirable to require soundness of dataflows, many of the systems with conditional behavior will not satisfy (iii). The problem is often not caused by the structure of the net, but by operations associated with transition labels that are being used. An appearance of a value that activates some part of the net may be dependent on the value with which the dataflow is initiated. Checking if the right value can appear would be undecidable as is determining if an NRC expression returns an empty set. Indeed, it is well known that NRC can simulate the relational algebra [9]. That is why we introduce a weaker *semi-soundness* notion without the third condition:

Definition 3.7.4 (semi-soundness)**.** A dataflow $D = \langle DFN, EN, TN, EA, PT\rangle$ with $DFN = \langle P, T, E, source, sink\rangle$ is semi-sound if and only if:

(i) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \xrightarrow{*} M$, if for some value $v'' : PT(sink)$ and history $h'' \in H$ it holds that $M(sink, \langle v'', h''\rangle) > 0$, then $M = output^D_{v''}$, and

(ii) for each value $v' : PT(source)$ and every marking M such that $input^D_{v'} \xrightarrow{*} M$ there exists a value $v'' : PT(sink)$ such that $M \xrightarrow{*} output^D_{v''}$.

3.7.1 Refinement rules

In this section we introduce refinement rules for generating what may be considered a well-structured dataflow. As we will show later, all dataflows generated in this way are semi-sound. By starting from a single place and applying the rules in a top-down manner we generate *blank dataflows* — dataflows without edge and transition naming. We call such generated blank dataflows *hierarchical blank dataflows*. From these we then obtain dataflows by adding edge and transition naming functions. These will be called *hierarchical dataflows*.

Definition 3.7.5 (blank dataflow)**.** A blank dataflow is a tuple $\langle DFN, EA\rangle$ where:

- $DFN = \langle P, T, E, source, sink\rangle$ is a dataflow net,
- $EA : (\circ T \to \{\text{“=true”}, \text{“=false”}, \text{“}=\emptyset\text{”}, \text{“}\neq\emptyset\text{”}, \text{“}*\text{”}, \varepsilon\}) \uplus (\circ P \to \{\text{“}*\text{”}, \varepsilon\})$ is an edge annotation function.

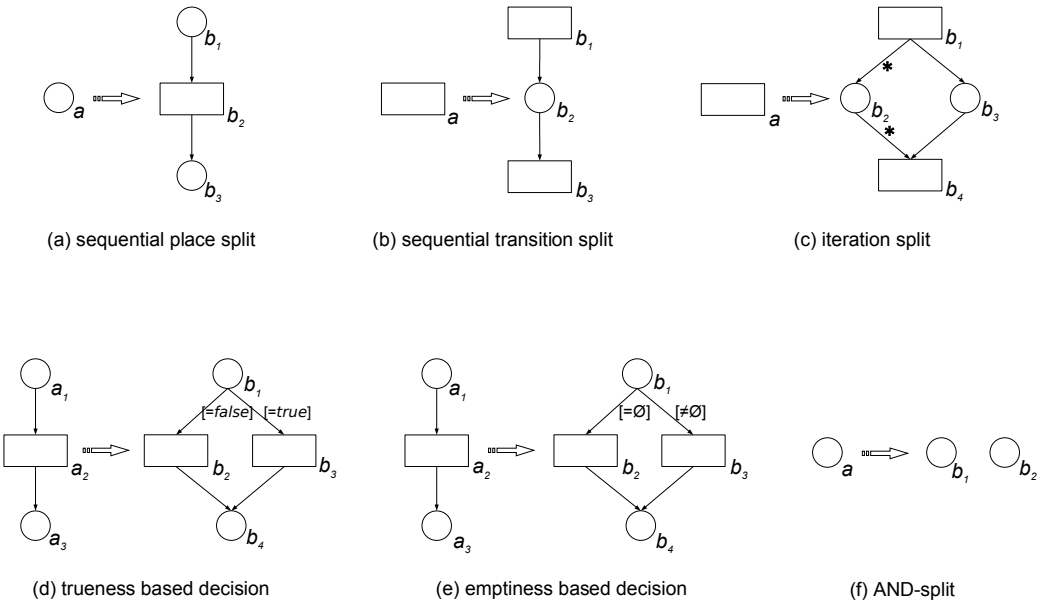

Figure 3.9: Refinement rules

The refinement rules are presented in Fig. 3.9. Each refinement replaces a subgraph presented on the lefthand side of the rule by the right-hand side one. The edge annotation for the replaced subgraph and the subgraph that it is replaced with is exactly as indicated. For each rule we define the concepts of *input nodes*, *output nodes* and *body nodes* as indicated in Table 3.2.

Table 3.2: Input, output and body nodes

	Rule a	**Rule b**	**Rule c**	**Rule d**	**Rule e**	**Rule f**
Input nodes	a, b_1	a, b_1	a, b_1	a_1, b_1	a_1, b_1	a, b_1, b_2
Output nodes	a, b_3	a, b_3	a, b_4	a_3, b_4	a_3, b_4	a, b_1, b_2
Body nodes	b_2	b_2	b_2, b_3	a_2, b_2, b_3	a_2, b_2, b_3	

The right-hand side subgraph is connected to the rest of the blank dataflow as follows:

- All the incoming edges of the left-hand side input node are reconnected

to all the input nodes of the right-hand side. The annotations are preserved. A visualization is presented in Fig. 3.10.

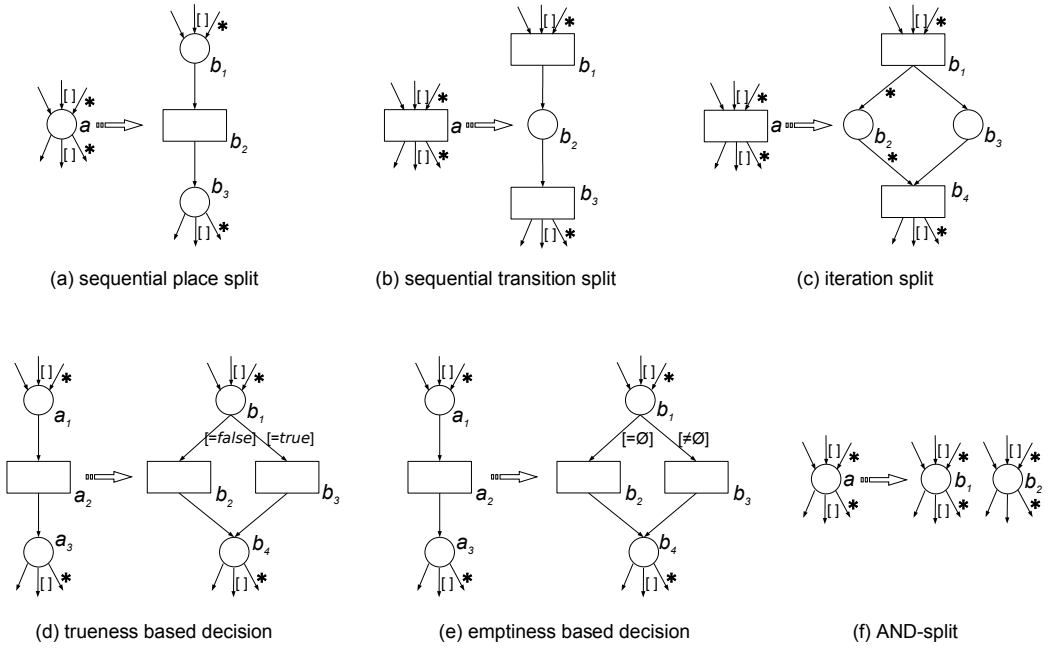

Figure 3.10: Reconnecting of subgraphs

- For rules d and e all the remaining, i.e., not shown in the rule, outgoing edges of the input nodes on the left-hand side are reconnected to the input nodes on the right-hand side. The annotations are preserved. A visualization is presented in Fig. 3.11.

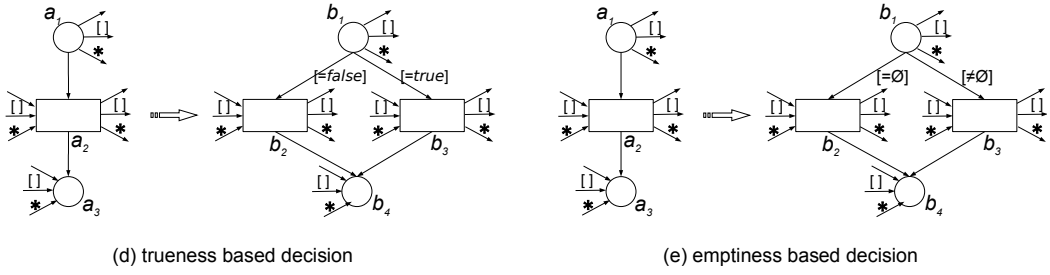

Figure 3.11: Reconnecting of subg. — additional edges for rules d and e

- All the outgoing edges of the left-hand side output node are reconnected to all the output nodes of the right-hand side. The annotations are preserved, with the exception that for rule f condition annotations are preserved only for outgoing edges of the node b_1 and outgoing edges of node b_2 are not annotated with conditions. A visualization is presented in Fig. 3.10.

- For rules d and e all the remaining, i.e., not shown in the rule, incoming edges of the output nodes on the left-hand side are reconnected to the output nodes on the right-hand side. The annotations are preserved. A visualization is presented in Fig. 3.11.

- All the incoming and outgoing edges of the left-hand side body nodes are reconnected to all the right-hand side body nodes. The annotations are preserved. A visualization is presented in Fig. 3.11.

There are certain preconditions that must hold when the rules are applied:

(i) For rule d to be applied, all the transitions in $\bullet a_1$ that are connected with a_1 by a non-annotated edge cannot have any unnest edges, i.e., for all $t \in \bullet a_1$ it holds that: if $EA(\langle t, a_1 \rangle) = \epsilon$, then for all $p \in t\bullet$ it holds that $EA(\langle t, p \rangle) \neq$ "$*$".

(ii) For rule d to be applied, all the transitions in $\bullet a_1$ cannot have any other output places that are connected by an edge annotated in the same way and on which an *emptiness based decision* is performed, i.e., for all $t \in \bullet a_1$ and for all $p \in t\bullet$ it holds that: if $p \neq a_1$ and $EA(\langle t, a_1 \rangle) = EA(\langle t, p \rangle)$, then for all $t' \in p\bullet$ it holds that $EA(\langle p, t' \rangle) \notin \{$"$=\emptyset$", "$\neq\emptyset$"$\}$.

(iii) For rule e to be applied, all the transitions in $\bullet a_1$ cannot have any other output places that are connected by an edge annotated in the same way and on which a *trueness based decision* is performed, i.e., for all $t \in \bullet a_1$ and for all $p \in t\bullet$ it holds that: if $p \neq a_1$ and $EA(\langle t, a_1 \rangle) = EA(\langle t, p \rangle)$, then it holds that for all $t' \in p\bullet$ it holds that $EA(\langle p, t' \rangle) \notin \{$"=true", "=false"$\}$.

(iv) For rule f to be applied, a has to have at least one incoming and one outgoing edge.

The first three preconditions are necessary so that it is always possible to label the generated blank dataflow such that it becomes a legal dataflow. (i) deals with a requirement that a token representing set value cannot be used to make a *trueness based decision* (see Fig. 3.12(i)), while (ii) and (iii) prevent using tokens with the same values in different kinds of tests (see Fig. 3.12(ii) and Fig. 3.12(iii)). Precondition (iv) guarantees that there is exactly one input and output place.

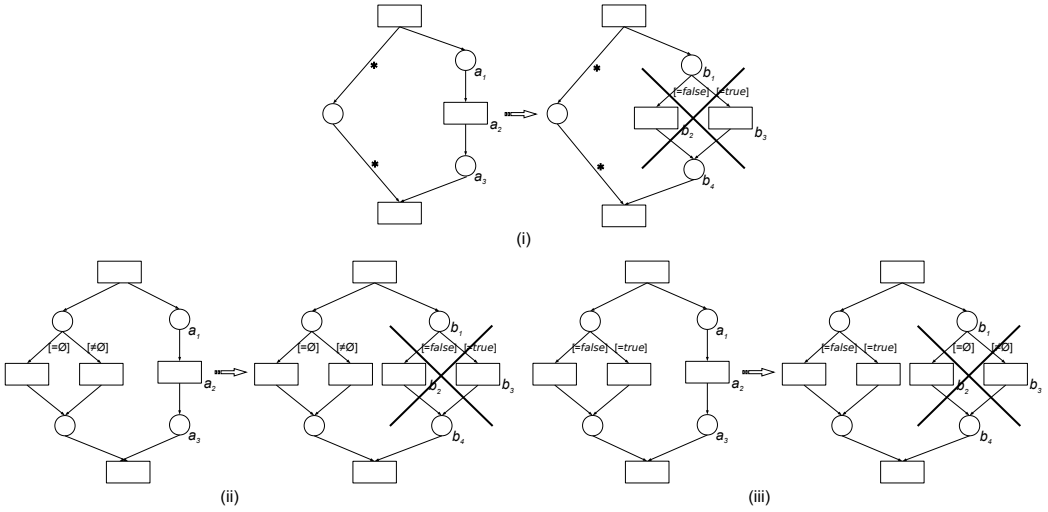

Figure 3.12: Preconditions

Definition 3.7.6 (hierarchical blank dataflow)**.** A blank dataflow which is obtained by starting with a blank dataflow that consists of a single place with no transitions and performing the transformations presented in Fig. 3.9 is called a hierarchical blank dataflow.

Definition 3.7.7 (hierarchical dataflow)**.** A hierarchical dataflow is a legal dataflow $D = \langle DFN, EN, TN, EA, PT \rangle$ obtained by labeling transitions and edges in a hierarchical blank dataflow $BDF = \langle DFN, EA \rangle$.

The rules and the aim to make dataflows structured as in structured programming languages were inspired by the work done on workflow nets by Chrząstowski-Wachtel et al. [13].

An instance of a computation of a particular dataflow, which starts in some input state, will be called a *run*. We will represent it as a pair of two

sequences. The first one will contain successive transitions that were fired and the second one subsequent states including the input state.

Definition 3.7.8 (run)**.** Let $D = \langle DFN, EN, TN, EA, PT\rangle$ be a dataflow with a dataflow net $DFN = \langle P, T, E, source, sink\rangle$. A sequence of transitions $t_1, \ldots, t_n \in T$ with a sequence of markings $M_0, \ldots, M_n$ of D, where M_0 is an input state, forms a run if and only if it holds that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$.

The run will be denoted as $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$. If M_n is an output state of D, then we will call such a run *complete*.

For a run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$, a place p and history h we define a delta of tokens in p after firing a given transition t_{i+1} in a state M_i:

$$\Delta_i(p, h) = \sum_{v \in CV} M_{i+1}(p, \langle v, h\rangle) - \sum_{v \in CV} M_i(p, \langle v, h\rangle)$$

We will also want to count tokens inserted to a place (since there are no cycles, during one transition tokens are never inserted to and consumed from a place at the same time):

$$\Delta_i^+(p, h) = \begin{cases} \Delta_i(p, h) & \text{if } \Delta_i(p, h) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The number of tokens with a given history h inserted into a place p during a run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$ will be called a *trace* of p and defined as $Tr(p, h) = \sum_{i=0}^{n-1} \Delta_i^+(p, h)$.

Lemma 3.7.9. *For each hierarchical dataflow $D = \langle DFN, EN, TN, EA, PT\rangle$ with a dataflow net $DFN = \langle P, T, E, source, sink\rangle$ and for each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$ of dataflow D, the trace of each place is bounded by* 1*, i.e., it holds that $\forall_{h \in H} \forall_{p \in P} Tr(p, h) \leq 1$.*

Theorem 3.7.10. *Every hierarchical dataflow is semi-sound.*

Proof. (of Lemma 3.7.9 and Theorem 3.7.10)
We will prove Lemma 3.7.9 and Theorem 3.7.10 together, by induction on the number of refinements applied in the generation of the blank dataflow. During this proof we will assume that in TL there are labels representing all the NRC expressions on the available external functions.

For a hierarchical dataflow consisting of only one place, all runs have empty transition sequence and the state sequence consists of only one state, which is an input and an output state at the same time. Therefore such dataflow is semi-sound and the sum in Lemma 3.7.9 contains no elements, thus is equal 0.

Let us assume by mathematical induction that for each hierarchical dataflow $D_n = \langle DFN_n, EN_n, TN_n, EA_n, PT_n \rangle$ with a dataflow net $DFN_n = \langle P_n, T_n,$
$E_n, source_n, sink_n \rangle$ whose hierarchical blank dataflow was generated in $n \geq 0$ refinements it holds that:

(1) for each run $M'_0 \xrightarrow{t_1} M'_1 \xrightarrow{t_2} \ldots \xrightarrow{t_d} M'_d$ of D_n every trace of every place is bounded by 1,

(2) for each value $v' : PT_n(source_n)$ and marking M' such that $input^{D_n}_{v'} \xrightarrow{*} M'$, if for some value $v'' : PT_n(sink_n)$ and history $h'' \in H$ it holds that $M'(sink_n, \langle v'', h'' \rangle) > 0$, then $M' = output^{D_n}_{v''}$, and

(3) for each value $v' : PT_n(source_n)$ and marking M' such that $input^{D_n}_{v'} \xrightarrow{*} M'$ there exists a value $v'' : PT_n(sink_n)$ such that $M' \xrightarrow{*} output^{D_n}_{v''}$.

We will show that if $D_{n+1} = \langle DFN_{n+1}, EN_{n+1}, TN_{n+1}, EA_{n+1}, PT_{n+1} \rangle$ with a dataflow net $DFN_{n+1} = \langle P_{n+1}, T_{n+1}, E_{n+1}, source_{n+1}, sink_{n+1} \rangle$ is an arbitrary hierarchical dataflow whose hierarchical blank dataflow was generated in $n+1$ refinements, then:

(i) for each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} M_m$ of D_{n+1} every trace of every place is bounded by 1,

(ii) for each value $v' : PT_{n+1}(source_{n+1})$ and each marking M such that $input^{D_{n+1}}_{v'} \xrightarrow{*} M$, if for some value $v'' : PT_{n+1}(sink_{n+1})$ and history $h'' \in H$ it holds that $M(sink_{n+1}, \langle v'', h'' \rangle) > 0$, then $M = output^{D_{n+1}}_{v''}$, and

(iii) for each value $v' : PT_{n+1}(source_{n+1})$ and each marking M such that $input^{D_{n+1}}_{v'} \xrightarrow{*} M$ there exists a value $v'' : PT_{n+1}(sink_{n+1})$ such that $M \xrightarrow{*} output^{D_{n+1}}_{v''}$.

Let us consider each possible case for the last, (n+1)st, refinement applied.

(a) The last applied refinement was a *sequential place split* (see Fig. 3.9a). Let $BDF_n = \langle DFN_n, EA_n \rangle$ with a dataflow net $DFN_n = \langle P_n, T_n, E_n, source_n, sink_n \rangle$ be a blank hierarchical dataflow generated by the first n refinements that generated the blank dataflow of D_{n+1}. Let $D_n = \langle DFN_n, EN_n, TN_n, EA_n, PT_n \rangle$ be a hierarchical dataflow labeled accordingly to the labeling of D_{n+1}. Since there is no b_2 transition in D_n, to keep D_n legal, the function that it computes is incorporated into the transitions that follow it directly, if there are any, or is omitted otherwise. That is $PT_n(a) = PT_{n+1}(b_1)$ and for each $t \in a\bullet$ it holds that

$$TN_n(t) = \begin{cases} TN_{n+1}(t)|^{\Phi_{TN_{n+1}(b_2)}}_{EN_{n+1}(\langle b_3,t\rangle)} & \text{if } EA(\langle a,t\rangle) \neq \text{“}*\text{”} \\ TN_{n+1}(t)|^{map(\Phi_{TN_{n+1}(b_2)})}_{EN_{n+1}(\langle b_3,t\rangle)} & \text{if } EA(\langle a,t\rangle) = \text{“}*\text{”} \end{cases}$$

Here $tl|^f_{l_i}$ means the transition label obtained from the transition label tl, by letting the input from edge l_i through f first[3]. Namely, if $IT(tl) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k \rangle$ and $f : \tau'_i \to \tau_i$, then $IT(tl|^f_{l_i}) = \langle l_1 : \tau_1, \ldots, l_i : \tau'_i, \ldots, l_k : \tau_k \rangle$, $OT(tl|^f_{l_i}) = OT(tl)$ and for all values $v_1, \ldots, v_k$ of the appropriate types $\Phi_{tl|^f_{l_i}}(\langle l_1 : v_1, \ldots, l_k : v_k \rangle) = \Phi_{tl}(\langle l_1 : v_1, \ldots, l_i : f(v_i), \ldots, l_k : v_k \rangle)$.

For each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} M_m$ of D_{n+1} we define a corresponding run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_d} M'_d$ of D_n. The transitions are fired in the same order, they consume the same tokens and functions produce the same results, but all occurrences of b_2 are omitted. It is easy to see that $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_d} M'_d$ is indeed a run of D_n and that it is unambiguously defined. Let us assume that the subsequence of not omitted transitions have indices $i_1, \ldots, i_d$. The markings of D_n are equal to their counterparts in D_{n+1} on all the places that appear in both of the dataflows (i.e. for every $p \in P_n \cap P_{n+1}$ and $k \in K$ it holds that $M_0(p,k) = M'_0(p,k)$ and $M_{i_1}(p,k) = M'_1(p,k)$, …, $M_{i_d}(p,k) = M'_d(p,k)$). Whereas place a contains all the tokens that in the counterpart marking are stored in b_1 as well as all the tokens

[3]Note that if two successive sequential place splits would be incorporated into the following transition over a “$*$” annotated edge, then the input from that edge would be preprocessed by $map(f) \circ map(g) = map(f \circ g)$, where f and g are functions of the removed b_2 transitions.

that were consumed from b_1 in order to produce the tokens that in the counterpart are stored in b_3. This correspondence in not an injection, though. For each run of D_n there can be many corresponding runs of D_{n+1}. This is because there is a choice when to fire b_2, if tokens inserted into a are not immediately consumed.

As for (i), the content of places in $M_0, M_{i_1}, \ldots, M_{i_d}$ is bounded by the content of places in $M'_0, M'_1, \ldots, M'_d$ respectively. In the remaining markings the only difference is that some tokens are consumed from b_1, processed by b_2 and the result is placed in b_3. Thus the traces of places in markings of D_{n+1} are limited by the traces of places in markings of D_n, for which the induction assumption holds.

As for (ii), we can assume without loss of generality that M_m is the first marking in $M_0, \ldots, M_m$ in which $sink_{n+1}$ is not empty. We will first consider the case when $sink_{n+1} \neq b_3$ and $t_m \neq b_2$. In M'_d of the corresponding run $sink_n$ is therefore not empty ($sink_n = sink_{n+1}$). From the induction assumption in M'_d there is only one token — the one in $sink_n$. Since in M_m there is the same number of tokens, then also in M_d there is only one token — the one in $sink_{n+1}$. In the case where $sink_{n+1} = b_3$, it is only possible for a token to be inserted into $sink_{n+1} = b_3$, when there was a token to be consumed from b_1. Yet, when the first token is inserted into b_1, there are no other tokens since in the corresponding run a token is inserted into a, which is a sink there. Since $M_0, \ldots, M_m$ was arbitrarily chosen, (ii) holds.

As for (iii), let $v' : PT_{n+1}(source_{n+1})$ and let M be a marking of D_{n+1} such that $input_{v'}^{D_{n+1}} \xrightarrow{*} M$. By the definition of marking reachability there exists a run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} M_m$, where $M_0 = input_{v'}^{D_{n+1}}$ and $M_m = M$. We know that for this run in D_n there exists a corresponding run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_d} M'_d$, where $M'_0 = input_{v'}^{D_n}$. From the semi-soundness of D_n it follows that for some $v'' : PT_n(sink_n)$ this corresponding run can be extended into a complete run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_d} M'_d \xrightarrow{t'_{d+1}} M'_{d+1} \xrightarrow{t'_{d+2}} \ldots \xrightarrow{t'_{d+q}} M'_{d+q}$, where $M'_{d+q} = output_{v''}^{D_n}$. For it in turn there exists a corresponding complete run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} M_m \xrightarrow{t_{m+1}} M_{m+1} \xrightarrow{t_{m+2}} \ldots \xrightarrow{t_{m+r}} M_{m+r}$ in D_{n+1}, which at the beginning is identical to the run of D_{n+1} we started from and in M_{m+r} place b_1 is empty (if $b_3 = sink_{n+1}$, b_1 can be emptied by firing b_2). This

completes the proof, since we have shown that $M_m \xrightarrow{*} output_{v'''}^{D_{n+1}}$, for $v''' = \begin{cases} \Phi_{TN_{n+1}(b_2)}(v'') & \text{if } a = sink_n \\ v'' & \text{otherwise.} \end{cases}$

(b) The last refinement was a *sequential transition split* (see Fig. 3.9b). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. In the resulting dataflow D_n, the label of a represents the composition of functions $\Phi_{TN_{n+1}(b_3)}$ and $\Phi_{TN_{n+1}(b_1)}$. That is $IT_n(TN_n(a)) = IT_{n+1}(TN_{n+1}(b_1))$, $OT_n(TN_n(a)) = OT_{n+1}(TN_{n+1}(b_3))$ and $\Phi_{TN_n(a)} = \Phi_{TN_{n+1}(b_3)} \circ \Phi_{TN_{n+1}(b_1)}$.

For each run $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} M_m$ of D_{n+1} we define a corresponding run $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_d} M'_d$ of D_n. The transitions are fired in the same order, they consume the same tokens and functions produce the same results, but all occurrences of b_1 are omitted and all occurrences of b_3 are replaced with a. It is easy to see that $M'_0 \xrightarrow{t'_1} M'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_d} M'_d$ is indeed a run of D_n and that it is unambiguously defined. Let us assume that the subsequence of not omitted (other that b_1) transitions have indices $i_1, \ldots, i_d$. The markings of D_n are equal to their counterparts in D_{n+1} on all the places that appear in both of the dataflows except the ones in $\bullet a = \bullet b_1$ (i.e. for every $p \in ((P_n \cap P_{n+1}) \setminus \bullet a)$ and $k \in K$ it holds that $M_0(p,k) = M'_0(p,k)$ and $M_{i_1}(p,k) = M'_1(p,k)$, $\ldots$, $M_{i_d}(p,k) = M'_d(p,k)$)). Whereas each place in $\bullet a$ contains all the tokens that in the counterpart marking are stored in the corresponding place in $\bullet b_1$ as well all the tokens that were consumed from that place in order to produce the tokens that are in the counterpart stored in b_2. This correspondence in not an injection, though. For each run of D_n there can be many corresponding runs of D_{n+1}. This is because there is a choice when to fire b_3, if tokens produced by a into $a\bullet$ are not immediately consumed.

The rest of the proof follows the one given for (a).

(c) The last refinement was an *iteration split* (see Fig. 3.9c). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. In the resulting dataflow D_n, the label of transition a represents a com-

position of three functions: $\Phi_{TN_{n+1}(b_4)}$, a *pair* function of appropriate type that constructs a pair of twice its argument, and a function $\Phi_{TN_{n+1}(b_1)}$. That is $IT_n(TN_n(a)) = IT_{n+1}(TN_{n+1}(b_1))$, $OT_n(TN_n(a)) = OT_{n+1}(TN_{n+1}(b_4))$ and $\Phi_{TN_n(a)} = \Phi_{TN_{n+1}(b_4)} \circ pair \circ \Phi_{TN_{n+1}(b_1)}$. The correspondence of runs is analogous as in (b). The rest of the proof follows.

(d) The last refinement was a *trueness based decision* (see Fig. 3.9d). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. That is $IT_n(TN_n(a_2)) = IT_{n+1}(TN_{n+1}(b_2)) = IT_{n+1}(TN_{n+1}(b_3))$ and $OT_n(TN_n(a_2)) = OT_{n+1}(TN_{n+1}(b_2)) = OT_{n+1}(TN_{n+1}(b_3))$, and for the edge names $EN_n(\langle a_1, a_2\rangle) = EN_{n+1}(\langle b_1, b_2\rangle) = EN_{n+1}(\langle b_1, b_3\rangle)$. Assume $IT_{n+1}(TN_{n+1}(b_2)) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, EN_n(\langle a_1, a_2\rangle) : PT(a_1)\rangle$. $TN_n(a_2)$ represents a function computing *if-then-else* expression that results in evaluating of either of $\Phi_{TN_{n+1}(b_2)}$ or $\Phi_{TN_{n+1}(b_3)}$. Which means that for every values $v_1, \ldots, v_k$ of appropriate types and every $v : PT(a_1)$ the result of function $\Phi_{TN_n(a_2)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_n(\langle a_1, a_2\rangle) : v\rangle)$ equals $\Phi_{TN(b_2)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_n(\langle a_1, a_2\rangle) : v\rangle)$, if $v =$ false, or $\Phi_{TN(b_3)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_n(\langle a_1, a_2\rangle) : v\rangle)$, otherwise.

The correspondence of runs in this case is a bijection. Transitions are fired in the same order, but all occurrences of b_2 and b_3 are replaced with a_2 or depending on the consumed token value a_2 is replaced by b_2 or b_3. The markings are equal to their counterparts in all the place that appear in both of the dataflows. Whereas a_1 contains the same tokens as b_1 and a_3 the same tokens as b_4.

The rest of the proof follows.

(e) The last refinement was an *emptiness based decision* (see Fig. 3.9e). The proof follows the one given for (d).

(f) The last refinement was an *AND-split* (see Fig. 3.9f). As previously, with the first n refinements, we can construct a blank hierarchical dataflow and label it accordingly to the labeling of D_{n+1}. Since *AND-split* was the last refinement applied in generation of blank dataflow of D_{n+1}, we know that $b_1\bullet = b_2\bullet$, $\bullet b_1 = \bullet b_2$ and thus $PT_{n+1}(b_1) = PT_{n+1}(b_2)$. For every transition $t_{n+1} \in b_1\bullet$, where $IT(TN_{n+1}(t_{n+1})) =$

$\langle l_1 : \tau_1, \ldots, l_k : \tau_k, EN_{n+1}(\langle b_1, t_{n+1}\rangle) : PT_{n+1}(b_1), EN_{n+1}(\langle b_2, t_{n+1}\rangle) : PT_{n+1}(b_2)\rangle$, its corresponding transition $t_n \in a\bullet$ is defined as follows:

- $IT(TN_n(t_n)) = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, EN_{n+1}(\langle b_1, t_{n+1}\rangle) : PT_{n+1}(b_1)\rangle$, that is $EN_n(\langle a, t_n\rangle) = EN_{n+1}\langle b_1, t_{n+1}\rangle$,
- $OT(TN_n(t_n)) = OT(TN_{n+1}(t_{n+1}))$,
- for all values $v_1, \ldots, v_k$ of appropriate types and all $v : PT_n$ the function computed by this transition is defined as follows

$$\Phi_{TN_n(t_n)}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_{n+1}(\langle b_1, t_{n+1}\rangle) : v\rangle) = \Phi_{TN_{n+1}(t_{n+1})}(\langle l_1 : v_1, \ldots, l_k : v_k, EN_{n+1}(\langle b_1, t_{n+1}\rangle) : v, EN_{n+1}(\langle b_2, t_{n+1}\rangle) : v\rangle)$$

The observation that in D_{n+1} places b_1 and b_2 get the same tokens as a gets in D_n completes the proof of Lemma 3.7.9.

The correspondence of runs in this case is a bijection. Transitions are fired in the same order. The markings are equal to their counterparts in all the place that appear in both of the dataflows. Whereas a contains the same tokens as b_1 and b_2, which have to have identical content because, each of the transitions consuming token from one of those places consumes a token with identical history from the other one ($b_1\bullet = b_2\bullet$) and from Lemma 3.7.9 we know that there is no choice of such tokens, so it must be exactly the one consumed from the first place. The rest of the proof follows.

□

3.7.2 The bioinformatics example revisited

We conjecture that in terms of expressible functions hierarchical dataflows are equivalent to NRC and thus, by following our claim in [21], are sufficient to describe most data-centric experiments in life sciences such as bioinformatics. To illustrate this we consider again the dataflow in Fig. 3.7. Closer inspection of this dataflow shows that it is not hierarchical. This is because the iterations in the dataflow start with a transition that only has unnesting edges as outgoing edges. This is in conflict with the *iteration split* rule in Fig. 3.9 which requires that next to the unnest-nest branch there is another branch

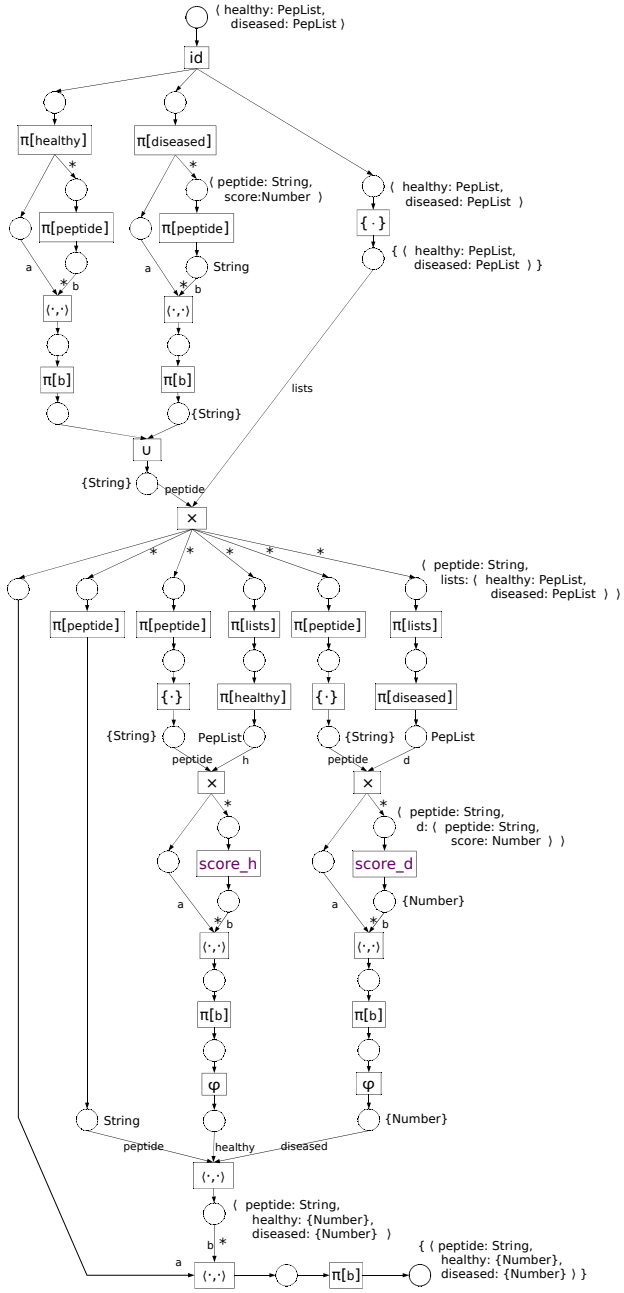

Figure 3.13: Finding differences in peptide content of two samples (hierarchical)

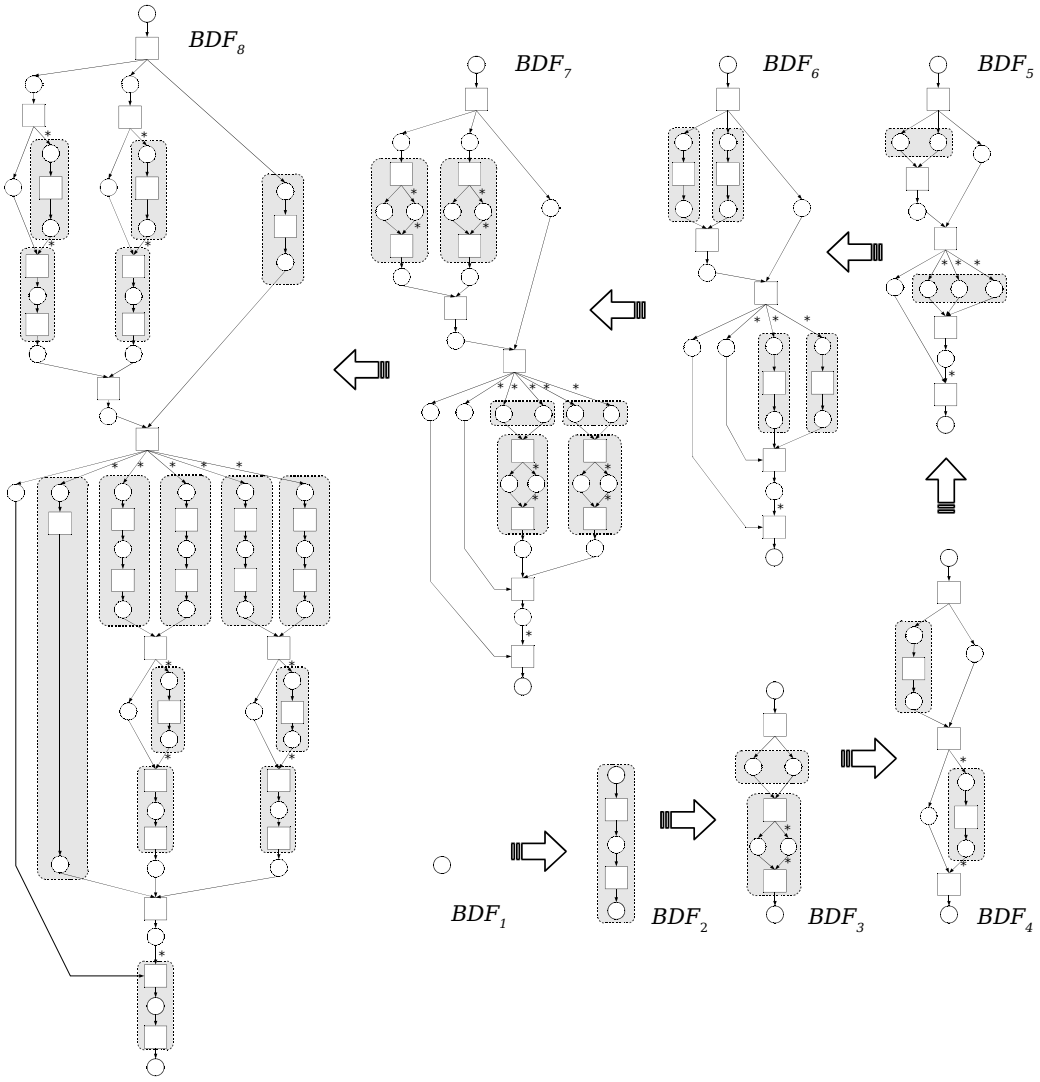

Figure 3.14: The generation of the blank dataflow from Fig. 3.7

that does not unnest and nest. Recall that the reason for this requirement is that if the function associated with the initial transition produces the empty set then the transition produces no tokens and the dataflow will probably not terminate properly. Observe that this is indeed what happens if the dataflow is presented with an empty "healthy" or "diseased" peptide list since the $\cup$ transition will never be enabled. The dataflow is therefore strictly speaking not semi-sound and cannot deal correctly with all possible input values. This soundness problem can be easily solved by introducing extra branches for the synchronization of the iterations as is shown in Fig. 3.13.

The corrected version of the dataflow can be shown to be hierarchical, which is demonstrated in Fig. 3.14 where the corresponding blank dataflow, called BDF_8 here, is generated from the blank dataflow with only one place, called BDF_1. The gray boxes indicate groups of nodes that were generated by expanding a single node in the preceding blank dataflow. For example, all nodes in BDF_2 where generated from the place in BDF_1 by applying the *sequential place split* and *sequential transition split*. For BDF_3 a place is split by using the *AND-split* and a transition is split by applying *iteration split*. In the following step BDF_4 is generated by applying the *sequential place split* to two places. Then BDF_5 is generated by using the *AND-split* for two places. Then for constructing BDF_6 some of the places that were just introduced are expanded with the *sequential place split*. In the next step BDF_7 is constructed by applying the *iteration split* to four transitions and the *AND-split* to two places. Finally, to construct BDF_7 the *sequential place split* and *sequential transition split* are applied several times.

As the preceding example shows, the hierarchical analysis of a dataflow can sometimes reveal subtle soundness problems. In the next section we present a tool with which for arbitrarily constructed dataflows it can be tested if they are hierarchical. The test is possible in polynomial time, similarly as for the rules proposed by Chrząstowski-Wachtel et al. for plain Petri nets [12].

3.8 DFL designer

In this section we present DFL designer — a COSW tool based on the DFL notation. Its main goal is to show that DFL can be used in practice as COSW specification language and that the techniques presented so far in this thesis are implementable. Additionally DFL designer aims to introduce to the COSW setting some useful features known from workflow modeling

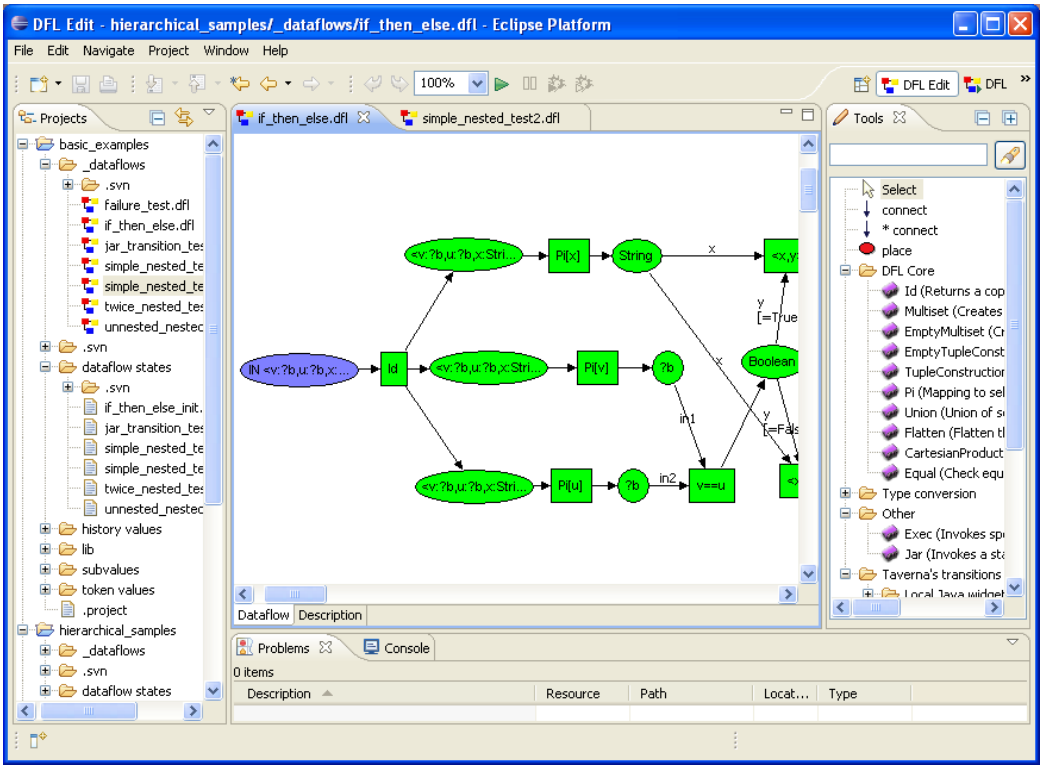

Figure 3.15: DFL designer — Edit perspective

and database tools.

DFL designer is developed as a plugin for the Eclipse platform [20] and is an open source project. It extends Eclipse with two new perspectives, DFL Edit (see Figure 3.15) and DFL Run (see Figure 3.16), which define the user interface configuration best suited for designing and enacting of dataflows respectively. Apart from the basic DFL operations DFL designer is provided with a huge library of bioinformatics services gathered by the Taverna workbench (see Section 1.2.1 and Chapter 2) which is also an open source project. Thanks to this it was possible to adapt already existing real life COSWs designed with Taverna and test on them the usefulness of DFL designer's novel features.

In this section we describe the features of DFL designer that are the most interesting and distinguishing from other COSW systems. We organize the features into three groups: correctness enforcement, enactment optimization, and debugging and testing support. Because of the space constraints and for

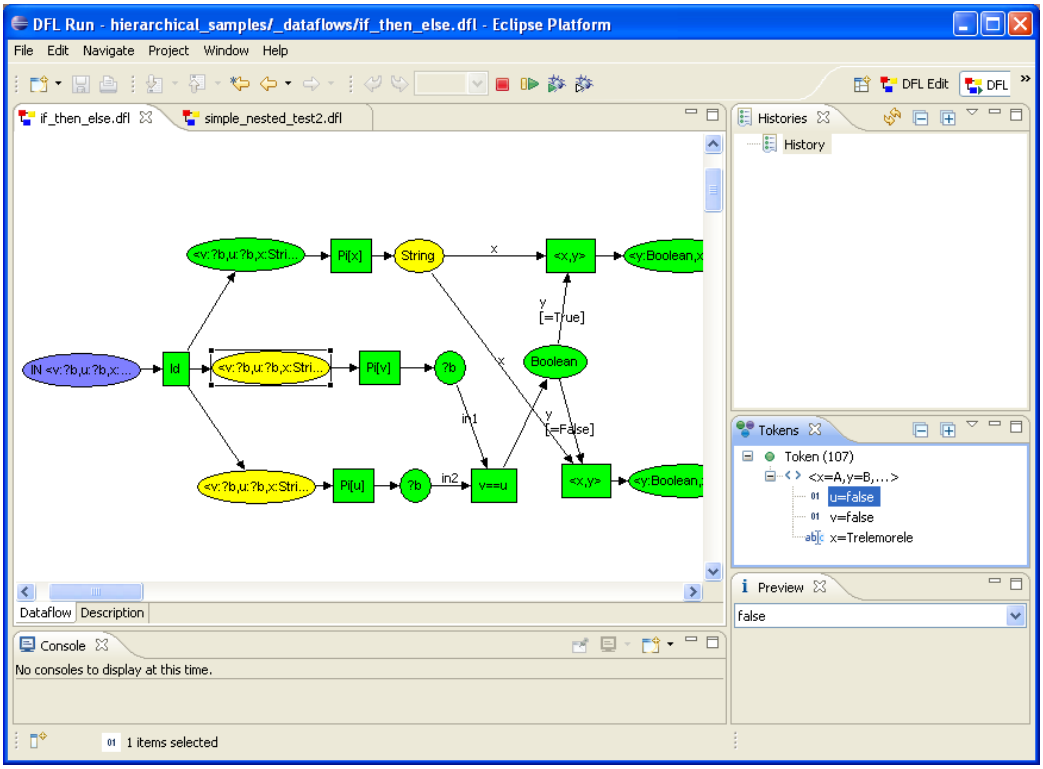

Figure 3.16: DFL designer — Run perspective

the accessibility of the presentation we use only abstract examples in this text. Real-life examples constructed by us or ported from Taverna are presented on the project web site at `http://code.google.com/p/dfldesigner`. The site also contains installation instructions and multimedia materials presenting DFL designer and its features.

3.8.1 Correctness enforcement

Source code editors in modern integrated development environments for strongly typed languages, like the one used in Eclipse to edit Java code, prevent the programmers from making certain types of errors by checking if operations are applied to parameters of correct types and if the result is assigned to a variable of a matching type. Thanks to the legality notion of DFL (see Definition 3.3.3) this type of aid is built-in in DFL designer and prevents the construction of illegal dataflows. While connecting the nodes on the diagram,

illegal constructions are prevented.

The enforcement of legality in the designer is more complex than for DFL itself. This is because in DFL the core operations are parametrized by types and labels, i.e., there are multiple instances of every operation with different input and output types. Yet, in the designer we have made those operations polymorphic to free the user from the inconvenience of dealing with many variants of the same operation. By doing this dataflows became polymorphic themselves and the types used in most cases cannot be determined exactly, but rather have to be expressed as patterns. Thus, while connecting the nodes on the diagram it is checked if corresponding patterns can be unified. For example, in the dataflow in Figure 3.17 copies of the input value are projected on fields u, v and x, so the input value has to be a record with at least such fields. Similarly, because the operations $f()$ and $g()$ consume and produce a string value, it also holds that string the type of field x. Checking if the combined restrictions can be met together with the computation of patterns for each place is a variant of the *type inference* problem [38, 49]. There are many results for type inference in industrial-strength functional programming languages like ML and Haskell. Some results are also available for a certain extension of NRC [65], but there the problem is proven to be NP-complete. Yet, for the polymorphic DFL the construction of a polynomial algorithm was possible thanks to: (1) a slightly different definition of the Cartesian product, which does not require that input values are sets of records with disjoint sets of field labels, and (2) a simple equality test for only arguments of the same basic type. A comprehensive discussion of which particular operations make the type inference problem NP-hard can be found in [67].

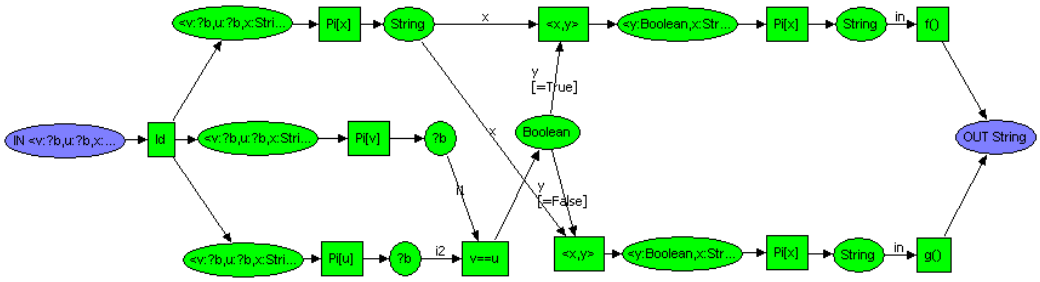

Figure 3.17: If-then-else example in DFL designer

As we have stressed in Section 3.7, the legality is not the only notion of correctness a dataflow author should keep in mind. Legal dataflows can

still follow a faulty design, e.g., continue the processing after the result has been produced, produce many results or not produce any results at all. We have called the class of dataflows that are correct in this way semi-sound and showed that it includes all hierarchical dataflows. In Sections 3.6 and 3.7.2 we have given an example of a real life dataflow which has some subtle flaws that are difficult to notice by a human reader. Those can be detected by checking the dataflow for hierarchicality and easily fixed by adhering to the refinement rules. Thus, apart from enforcement of legality, DFL designer can verify if a dataflow is hierarchical (see Section 3.8.2 for the details of the algorithm), which is an interesting example of how Petri net structural analysis can be applied to aid COSW practitioners.

3.8.2 Enactment optimization

Many tricks can be usually applied before executing a program, e.g., while compiling or interpreting it, to speed up its execution. Optimization techniques are also developed, applied and studied in the context of COSWs enactment, but most often this research concentrates on aspects shared with business workflows and grid techniques, e.g., which of many semantically equivalent services to choose [29] or how to execute the COSW on distributed resources such as the Grid [16]. Since we emphasize the data processing aspects of COSWs, we are more interested in application of the query optimization results like the ones for NRC. It should be noted here, that the most effective optimization results for query languages in general and for NRC in particular depend upon algebraic identities that only hold if the involved operations are side-effect free. It is our observation that most operations used in COSWs are indeed side-effect free. Also the basic operators for which these algebraic identities are known, like products, are present in COSW languages, sometimes explicitly as in DFL and sometimes implicitly as in Taverna, so these identities can also be used here for optimization.

The application of the query optimization results to COSWs can be done by either developing similar results for COSW specification languages or by translating COSWs to some query language for which optimized execution engines exist. As we show with DFL designer, for hierarchical dataflows a mapping to NRC is possible. Also NRC has many stable execution engines [70], though by the time of this writing we have implemented the mapping algorithm but have not integrated DFL designer with an NRC execution engine yet. The mapping is achieved by an extension of the algorithm that

checks if a dataflow is hierarchical. Such a check is done by determining if a reversal of the refinements from Figure 3.9 is possible until only one transition with a source and a sink place is left. Only if it is, the dataflow is hierarchical. During the merging of dataflow nodes according to the reversed refinement rules, operations represented by transitions can be composed, so that in each step the dataflow as a whole computes the same function. The idea for the compositions of functions is presented on Figure 3.18.

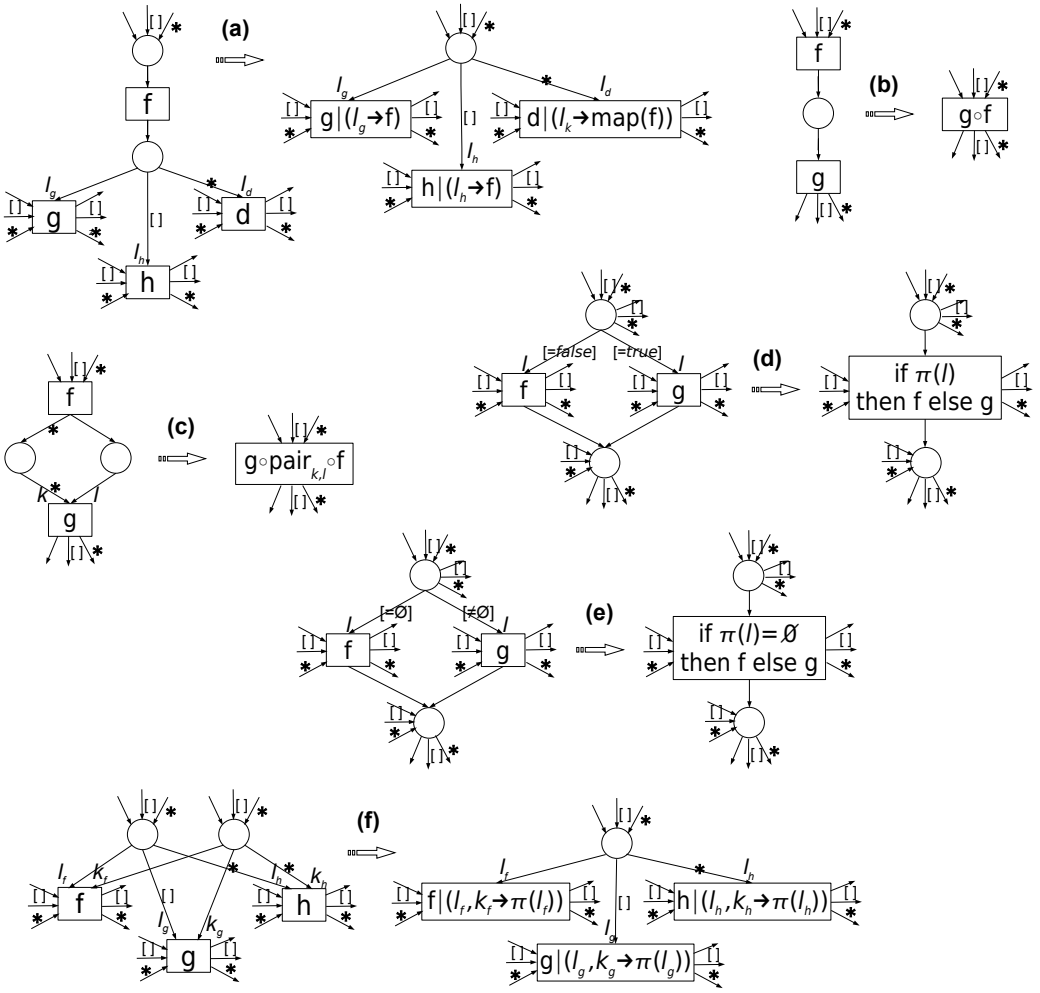

Figure 3.18: Composition of functions while reversing the refinements

For example, in the case of the iteration split of rule (c), on left hand side, the g function computed by the lower transition is provided on both of

its arguments with the result of function f which is computed by the upper transition. Thus, the transition on the right hand side which represents their combination computes the function $g \circ pair_{k,l} \circ f$, where $pair_{k,l}$ returns a record with field labels k, l and the same value on both of them, i.e., the input value. Here the "$*$" annotated edges are unimportant since no processing occurs between the unnesting and nesting. If a structural recursion indeed takes place in the dataflow it is incorporated into the calculated functions by the reversal of the sequential place split of rule (a). The notation $d|(l_d \rightarrow map(f))$ represents a modification of function d such that the value provided on the label l_d is first processed by the function $map(f)$[4]. The $map(f)$ in turn is a fundamental NRC operation that expects a collection and processes all of its elements by f.

The presentation of the idea of the mapping algorithm in an algebraic way, i.e., with function composition, was given here only because of its compactness. The actual algorithm generates a calculus like expression. For the if-then-else example from Figure 3.17 it produces:

$$
\begin{array}{l}
\textbf{if } (id(\$"IN").u == id(\$"IN").v) \textbf{ then} \\
\quad f(in : \langle y : (id(\$"IN").u == id(\$"IN").v), \\
\qquad\quad x : id(\$"IN").x \\
\qquad\quad \rangle.x) \\
\textbf{else} \\
\quad g(in : \langle x : id(\$"IN").x, \\
\qquad\quad y : (id(\$"IN").u == id(\$"IN").v) \\
\qquad\quad \rangle.x)
\end{array}
$$

This expression can and would be simplified by the NRC optimizer to **if** $(\$"IN".u == \$"IN".v)$ **then** $f(\$"IN".x)$ **else** $g(\$"IN".x)$ by removing the identity operations and noticing that records are constructed just to be immediately projected on one of their fields. This extra complexity is not an indication of bad design of DFL but of its graphical nature. Some constructions, e.g., conditionals, filters and inner joins, are difficult to express in a graphical notation where values cannot be accessed by context, but all

[4]Similarly (see case (f)), the notation $f|(l_f, k_f \rightarrow \pi[l_f])$ represents a modification of function f such that the values provided on the labels l_f and k_f are first processed by the function $\pi[l_f]$.

flow of data has to be presented explicitly with an edge in the diagram. This does not change even with the use of high level, user friendly graphical notations that include many implicit features and a lot of syntactic sugar, such as the one of the Taverna workbench.

3.8.3 Debugging and testing COSWs with interactive firing of transitions

DFL designer can execute dataflows interactively, where the user selects which of the enabled transitions will fire next. This is known from Petri net tools as the *token game*. Yet, since the tokens in DFL carry data values, an extra feature is present that allows to disable arbitrary tokens and thus determine not only which transition will fire, but also which data values it will consume. For user convenience it can be chosen whether the new tokens are produced as enabled or disabled. At every moment of the interactive execution the user has full control over the distribution of tokens and their values. It is possible to inspect and edit what tokens are in each place, what are their unnesting histories and what values they transport. The state of all places can be saved to and loaded from an XML file. Also the information about every single token, its data value and history can be stored this way.

The interactive firing of transitions together with the full control over the state, allows the user to precisely understand the execution semantics of the defined dataflow and gives the means to effectively debug and test them. In particular the user can: (1) check what values are returned by transitions and make sure that services represented by them behave as expected, (2) repeat the experiments from saved partial states, (3) enforce and test all possible variants of execution which is especially useful when complex synchronization protocol is being defined, and (4) experiment in a "what would happen if" way, by redefining the state during the execution.

3.8.4 Further research

There are many interesting directions of extending DFL designer and continuing our research. As we have already shown in [56] that an XQuery [59] (a standard query language for XML data) engine can be integrated with a COSW workbench in such a way that parts of the workflow can be expressed as a query and this query can access the operations provided by the workbench. Similar integration with an NRC or XQuery engine is a natural next

step in the development of DFL designer. It would also allow to enact the dataflow more effectively since hierarchical fragments of dataflows which use only side-effect free operations could be automatically translated into the query language and optimized. Another interesting topic is adding provenance support [27], so that information about how each data item has been computed, i.e., by which services and from what input values, is collected. This is important for scientists using COSWs for knowing how much they can rely on the data, i.e., if it is very reliable data obtained by direct observation in laboratory or less reliable data resulting from several approximation algorithms applied subsequently. Finally, further analysis algorithms can be designed for DFL by following the results available for Petri nets. For example, there are interesting results by Piotr Chrząstowski-Wachtel [11] showing that token distributions can also be checked for soundness, i.e., whether starting from such a marking the computation can always be correctly completed regardless of how it proceeds.

Apart from testing new ideas and setting new directions DFL designer can be further developed in two ways: as a general use COSW tool and as a supplement for other existing systems. The first requires additional work on user friendliness, support of further domain specific operations and enriching the DFL notation with syntactic sugar and implicit features like an implicit iteration mechanism. The second requires the definition of a mapping from notations used in other systems, e.g., from Scufl, to DFL such that COSWs defined in other tools could be automatically translated to DFL to take advantage of the analysis algorithms that are available for it. This way DFL could be used as a formal core of COSW languages similarly as NRC is a formal core of object-oriented and hierarchical query languages.

Chapter 4

Summary of the presented results and further research

4.1 Summary

In this thesis we have investigated and formalized the semantics of Scufl — the COSW specification language of Taverna workbench and proved some basic properties of all COSWs defined in Scufl.

Because the complexities of Taverna have convinced us that it is important to have a cleaner model, we have proposed a new hybrid formal model for specification of COSWs from first principles, that combines Petri nets and NRC. We have also shown that results available for Petri nets can be adapted to DFL.

In addition, we have constructed DFL designer — a new COSW system that is based on the DFL notation, incorporates services available in Taverna workbench and provides some features, unique in COSW setting, like correctness enforcement, possible enactment optimization through NRC, and debugging and testing support.

4.2 Publications and related research

The results presented in this thesis have been published or are under submission in several papers with several different coauthors. We survey these publications here.

The results from Chapter 2 are under submission to the Fundamenta

Informaticae journal. The material from Chapter 3, but without Section 3.8, has been presented in [28, 21]. We also intend to present DFL designer, which is described in Section 3.8, at some conference or workshop.

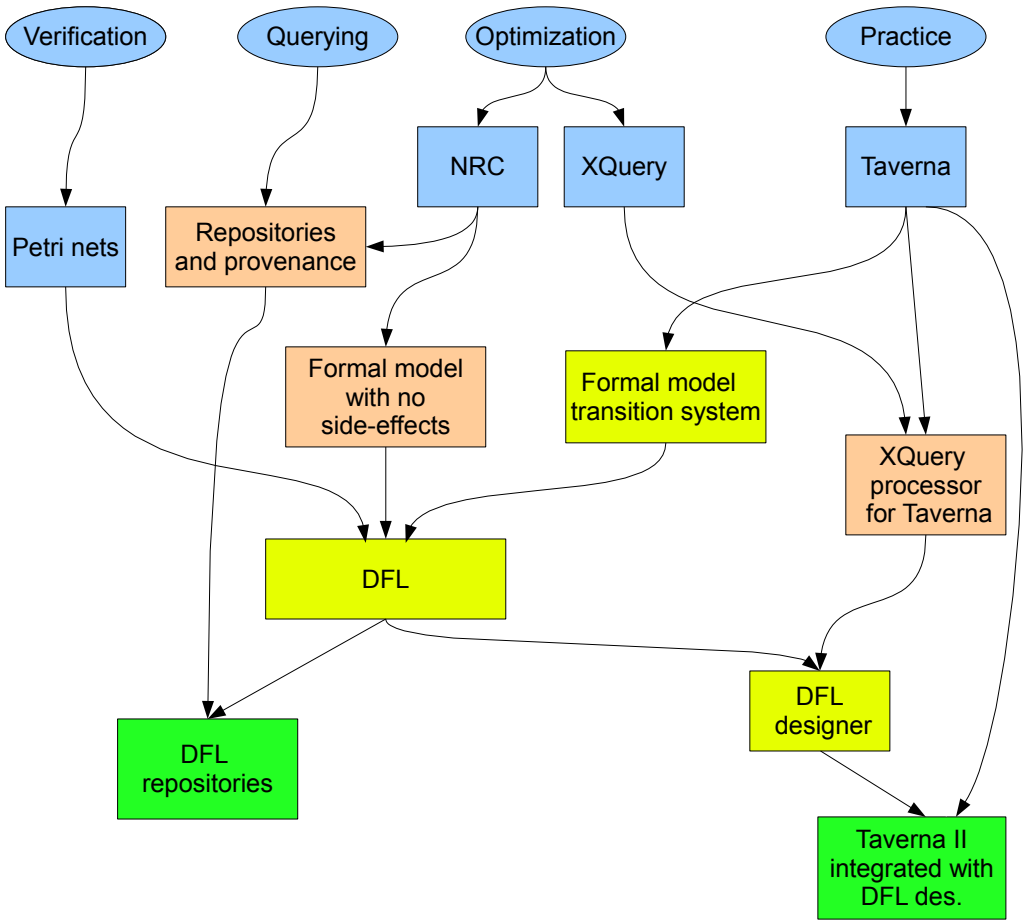

Figure 4.1: Interdependencies and context of the results presented in this thesis

The described results are part of a more extensive research on related topics, which also have lead to several publications of ours or in which we were involved, but which are only referenced in the text. In Fig. 4.1 we outline the relationships between this other work and the work presented in this thesis. We use blue rectangles to represent already existing results, pink rectangles to represent results of ours which were only referenced in this

thesis, yellow rectangles are the main results covered in this thesis and green rectangles are interesting further research areas. The arrows symbolize the direction of influence.

We have done some work on the importance of side-effects in COSWs and the possibility to specify COSWs with just NRC [22, 21]. We have also extended Taverna with the possibility to execute XQuery queries and specify parts of COSW with XQuery [56]. Finally, we have defined a formal model of COSW repositories [27].

In addition we have also proposed and tested in practice the idea of using a spreadsheet as a COSW specification interface and execution engine. For that, in [18], we have extended the Calc spreadsheet from the OpenOffice.org [47] package with the most important features of Taverna.

4.3 Further research

An interesting continuation of this research is to use DFL as a base model for the COSW notations found in popular tools. This way the formal methods developed for DFL would immediately become available for other COSW languages. For example, a mapping from Scufl to DFL could be created. We have already partially investigated such a mapping for the clean core of Scufl as it is defined, by Turi et al. in [60]. In fact, as we claim in [21], if side effects are not considered, as in Turi et al., then NRC itself is enough. A mapping of the whole of Scufl may be more difficult, since in Scufl all the threads of iteration over a nested Scufl graph start from a clean initial state. Expressing this in DFL would require additional garbage token cleaning constructs. Yet, as we know from our contacts with Taverna team, in the next version of the system the select-first incoming-links strategy and failure mechanism, whose simulation in DFL would cause garbage tokens to appear, will be modified.

Appendix A

Basic properties of lexicographical ordering of number vectors

Definition A.0.1 (Number vector)**.** A *number vector of dimension* $n \in \mathbb{N}$ is a tuple $\bar{c} = (c_1, \ldots, c_n)$ with $c_1, \ldots, c_n \in \mathbb{N}$. The set of all such vectors is denoted as $\mathbb{N}^n$.

If $\bar{c} = (c_1, \ldots, c_n)$, then we let $(c_0, \bar{c})$ denote the number vector $(c_0, c_1, \ldots, c_n)$.

Definition A.0.2 (Number vector ordering)**.** Over $\mathbb{N}^n$ we let $\leq^n$ denote the lexicographical ordering, i.e., it holds that:

(i) $() \leq^0 ()$, and

(ii) $(c_1, \bar{c}) \leq^{n+1} (d_1, \bar{d})$ iff (a) $c_1 < d_1$ or (b) $c_1 = d_1$ and $\bar{c} \leq^n \bar{d}$.

Proposition A.0.3. $(\mathbb{N}^n, \leq^n)$ *is a partially ordered set for each* $n \in \mathbb{N}$.

Proof. We show this by induction on n. For $n = 0$ this is clear since $\mathbb{N}^0 = \{()\}$ and $() \leq^0 ()$. For $n + 1$ we can show, using the induction assumption for n, the reflexivity, antisymmetry and transitivity as follows:

Reflexivity Let $(c_1, \bar{c})$ be an arbitrary vector from $\mathbb{N}^{n+1}$. By induction we know that $\bar{c} \leq^n \bar{c}$ and by (2b) it then follows that $(c_1, \bar{c}) \leq^{n+1} (c_1, \bar{c})$.

Antisymmetry Let $\bar{c}', \bar{d}' \in \mathbb{N}^{n+1}$ such that $\bar{c}' \leq^{n+1} \bar{d}'$ and $\bar{d}' \leq^{n+1} \bar{c}'$ where $\bar{c}' = (c_1, \bar{c})$ and $\bar{d}' = (d_1, \bar{d})$. Based on the definition of number vector ordering this is only possible if $c_1 \leq d_1$ and at the same time $d_1 \leq c_1$ which together imply $c_1 = d_1$. From this and the $\bar{c}' \leq^{n+1} \bar{d}'$ it follows that $\bar{c} \leq^n \bar{d}$. Similarly we get $\bar{d} \leq^n \bar{c}$. Now from the induction assumption we know that $\bar{c} = \bar{d}$ which completes the proof since we already showed that $c_1 = d_1$.

Transitivity Assume that $(c_1, \bar{c}) \leq^{n+1} (d_1, \bar{d})$ and that $(d_1, \bar{d}) \leq^{n+1} (e_1, \bar{e})$. Then one of the following cases holds: (i) $c_1 < d_1$ and $d_1 < e_1$, (ii) $c_1 < d_1$ and $d_1 = e_1$, (iii) $c_1 = d_1$ and $d_1 < e_1$, and (iv) $c_1 = d_1 = e_1$, $\bar{c} \leq^n \bar{d}$ and $\bar{d} \leq^n \bar{e}$. In the first three cases (i), (ii) and (iii) it follows that $c_1 < e_1$, and therefore $(c_1, \bar{c}) \leq^{n+1} (e_1, \bar{e})$. In case (iv) it follows that $c_1 = e_1$ and by induction that $\bar{c} \leq^n \bar{e}$, and therefore $(c_1, \bar{c}) \leq^{n+1} (e_1, \bar{e})$.

□

Definition A.0.4 (Well-founded)**.** A partially ordered set $(V, \leq_V)$ is said to be *well-founded* if it holds for every non-empty subset $V' \subseteq V$ that it contains at least one minimal element, i.e., an element $v \in V'$ such that for all $w \in V'$ if $w \leq_V v$ then $w = v$.

Proposition A.0.5. *The partial order $(\mathbb{N}^n, \leq^n)$ is well-founded.*

Proof. We prove this with induction on n. For $n = 0$ it holds since there is only one element in $\mathbb{N}^0$. Next, we consider $n + 1$. Let c_1' be the smallest number in $\{c_1 \mid (c_1, \bar{c}) \in \mathbb{N}^{n+1}\}$. Then let $\bar{c}'$ be the minimal element in $\{\bar{c} \mid (c_1', \bar{c}) \in \mathbb{N}^{n+1}\}$ w.r.t. $\leq^n$, which by induction exists. We now show that $(c_1', \bar{c}')$ is a minimal element of $\mathbb{N}^{n+1}$. Assume that $(c_1, \bar{c}) \leq^{n+1} (c_1', \bar{c}')$. By the definition of $\leq^{n+1}$ it holds that $c_1 \leq c_1'$ and by the definition of c_1' that $c_1' \leq c_1$, and so $c_1 = c_1'$. From this it follows that $\bar{c} \leq^n \bar{c}'$. By induction and the fact that $\bar{c}'$ is a minimal element of a set of which $\bar{c}$ is also an element, it follows that $\bar{c}' = \bar{c}$. It therefore holds that $(c_1, \bar{c}) = (c_1', \bar{c}')$. □

Proposition A.0.6. *$(V, \leq_V)$ is well-founded iff V contains no infinite descending chains, i.e., there exists no injective function $f : \mathbb{N} \to V$ such that for every $n \in \mathbb{N}$ it holds $f(n + 1) \leq_V f(n)$.*

Proof. We will prove both implications by contradiction. Let $V' \subseteq V$ be a non-empty subset without a minimal element. The sequence $f : \mathbb{N} \to V'$ can be defined as follows. $f(0)$ is an arbitrary element of V'. If f is defined for all

$k \leq n$ and for every $k < n$ it holds $f(k+1) \leq_V f(k)$ but $f(k+1) \neq f(k)$ then as $f(n+1)$ we choose any other element from V' such that $f(n+1) \leq_V f(n)$. The existence of such element follows from the fact that $f(n)$ is not minimal in V'. The other way around, if there exists an infinite descending chain f, then the image $\{f(1), f(2), \ldots\}$ is a non-empty subset of V that has no minimal element. □

Corollary A.0.7. *The partially ordered set* $(\mathbb{N}^n, \leq^n)$ *contains no infinite descending chains.*

Proof. This follows directly from Propositions A.0.5 and A.0.6. □

Bibliography

[1] W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[2] Anastassia Ailamaki, Yannis E. Ioannidis, and Miron Livny. Scientific workflow management by database management. In *Statistical and Scientific Database Management*, pages 190–199, 1998.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, 1990.

[4] Herman Balsters and Maarten M. Fokkinga. Subtyping can have a simple semantics. *Theor. Comput. Sci.*, 87(1):81–96, 1991.

[5] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 343–354. ACM, 2006.

[6] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. GenBank. *Nucleic Acids Res*, 36(Database issue), January 2008.

[7] Gèrard Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, covers the 6th European Workshop on Applications and Theory in Petri Nets-selected papers*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40, London, UK, 1986. Springer-Verlag.

[8] B. Boeckmann, A. Bairoch, R. Apweiler, MC. Blatter, A. Estreicher, and et al. The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic Acids Research*, 31:365–370, 2003.

[9] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[10] Vassilis Christophides, Richard Hull, and Akhil Kumar. Querying and splicing of XML workflows. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 386–402, London, UK, 2001. Springer-Verlag.

[11] Piotr Chrząstowski-Wachtel. Determining sound markings in structured nets. *Fundamenta Informaticae*, 72(1-3):65–79, 2006.

[12] Piotr Chrząstowski-Wachtel. private communication, 2007.

[13] Piotr Chrząstowski-Wachtel, Boualem Benatallah, Rachid Hamadi, Milton O'Dell, and Adi Susanto. A top-down Petri net-based approach for dynamic workflow modeling. In Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2003.

[14] Susan B. Davidson, G. Christian Overton, Val Tannen, and Limsoon Wong. BioKleisli: A Digital Library for Biomedical Researchers. *Int. J. on Digital Libraries*, 1(1):36–53, 1997.

[15] E. Deelman, G. Singh, Mei-Hui Su, J. Blythe, Y. Gil, C Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[16] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[17] UC Berkeley Department of EECS. Ptolemy II project and system. `http://ptolemy.eecs.berkeley.edu/ptolemyII`, 2008.

[18] Marek Dopiera, Adam Kawa, Piotr Krewski, Jacek Sroka, Jerzy Tyszkiewicz, and Tomek Weksej. Tavernalc: How to transform your OpenOffice Calc into a grid. In *OpenOffice.org Conference (OOoCon)*, 2007.

[19] D. Dumont, J.P. Noben, J. Raus, P. Stinissen, and J. Robben. Proteomic analysis of cerebrospinal fluid from multiple sclerosis patients. *Proteomics*, 4(7), 2004.

[20] Eclipse — an open development platform. `http://www.eclipse.org`.

[21] Anna Gambin, Jan Hidders, Natalia Kwasnikowska, Sławomir Lasota, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. NRC as a formal model for expressing bioinformatics workflows. Poster at ISMB, 2005. Poster.

[22] Anna Gambin, Jan Hidders, Natalia Kwasnikowska, Sławomir Lasota, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. Well-constructed workflows in bioinformatics. In *Workshop on Database Issues in Biological Databases (DBiBD)*, 2005.

[23] C. A. Goble and D. C. De Roure. myExperiment: social networking for workflow-using e-scientists. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 1–2, New York, NY, USA, 2007. ACM Press.

[24] Grid workflow forum. `http://www.gridworkflow.org/snips/gridworkflow/space/Projects`.

[25] Adnene Guabtni and François Charoy. Multiple instantiation in a dynamic workflow environment. In Anne Persson and Janis Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[26] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. Petri net + nested relational calculus = dataflow. In *OTM Conferences (1)*, pages 220–237, 2005.

[27] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In *Proc. of the 4th Int. Workshop on Data Integration in Life Sciences (DILS)*, volume 4544/2007 of *LNBI*, pages 105–121, Philadelphia, PA, USA, June 27–29 2007.

[28] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. DFL: A dataflow language based on Petri nets and nested relational calculus. *Information Systems*, 33(3):261–284, 2008.

[29] Lican Huang, Asif Akram, Rob Allan, David W. Walker, Omer F. Rana, and Yan Huang. A workflow portal supporting multi-language interoperation and optimization: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(12):1583–1595, 2007.

[30] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucl. Acids Res.*, 34:W729–732, 2006.

[31] Kurt Jensen. *Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volumes 1 and 2*. Springer-Verlag, London, UK, 1996.

[32] C. Johnson and S. Parker. Applications in computational medicine using SCIRun: a computational steering programming environment, 1995.

[33] G. Kandaswamy, L. Fang, Y. Huang, S. Shirasuna, S. Marru, and D. Gannon. Building web services for scientific grid applications. *IBM Journal of Research and Development*, 50(2/3):249–260, 2006.

[34] P. Li, K. Hayward, C. Jennings, K. Owen, T. Oinn, R. Stevens, S. Pearce, and A. Wipat. Association of variations in I kappa B-epsilon with GravesŠ disease using classical and myGrid methodologies. In *Proceedings of the UK e-Science All Hands Meeting 2004*, Nottingham, UK, September 2004.

[35] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Sci-

entific workflow management and the Kepler system: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.

[36] Shalil Majithia, Matthew S. Shields, Ian J. Taylor, and Ian Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 514–524. IEEE Computer Society, 2004.

[37] Timothy M. McPhillips, Shawn Bowers, and Bertram Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In Ulf Leser, Felix Naumann, and Barbara A. Eckman, editors, *DILS*, volume 4075 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2006.

[38] John C. Mitchell. *Foundations of programming languages*. MIT Press, Cambridge, MA, USA, 1996.

[39] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[40] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[41] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[42] National Center for Biotechnology Information. NCBI Blast. `http://www.ncbi.nlm.nih.gov/blast/Blast.cgi`.

[43] Andreas Oberweis and Peter Sander. Information system behavior specification by high level Petri nets. *ACM Trans. Inf. Syst.*, 14(4):380–420, 1996.

[44] Object Management Group. Unified modeling language resource page. `http://www.uml.org`.

[45] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[46] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1067–1100, 2006.

[47] OpenOffice.org — the free and open productivity suite. `http://www.openoffice.org`.

[48] Cesare Pautasso and Gustavo Alonso. The JOpera visual composition language. *Journal of Visual Languages and Computing (JVLC)*, 16:119–152, 2005.

[49] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[50] Hajo A. Reijers. *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. Number 2617 in Lecture Notes in Computer Science. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[51] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[52] P. Rice, I. Longden, and A. Bleasby. EMBOSS: the European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, June 2000.

[53] Peter M. Rice, Alan J. Bleasby, Syed A. Haider, Jon C. Ison, Shaun McGlinchey, and Mahmut Uludag. EMBRACE: Bioinformatics Data and Analysis Tool Services for e-Science. *e-science*, 0:146, 2006.

[54] A. Rowe, D. Kalaitzopoulos, M. Osmond, M. Ghanem, and Y. Guo. The discovery net system for high throughput bioinformatics. In *ISMB (Supplement of Bioinformatics)*, pages 225–231, 2003.

[55] Scientific workflows survey. `http://www.extreme.indiana.edu/swf-survey`.

[56] Jacek Sroka, Grzegorz Kaczor, Jerzy Tyszkiewicz, and Andrzej M. Kierzek. XQTav: an XQuery processor for Taverna environment. *Bioinformatics*, 22(10):1280–1281, May 2006.

[57] R. Stevens, H.J. Tipney, C. Wroe, T. Oinn, M. Senger, C.A. Goble, P. Lord, A. Brass, and M. Tassabehji. Exploring Williams-Beuren syndrome using ^{my}Grid. In *Proceedings of 12th International Conference on Intelligent Systems in Molecular Biology*, 2004.

[58] Dan Suciu and Limsoon Wong. On two forms of structural recursion. In *ICDT*, pages 111–124, 1995.

[59] The World Wide Web Consortium. XML query working group public page. `http://www.w3.org/XML/Query`.

[60] Daniele Turi, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn. Taverna workflows: Syntax and semantics. In *e-Science and Grid Computing, IEEE International Conference on*, pages 441–448, 2007.

[61] University of Virginia. FASTA Sequence Comparison. `http://wrpmg5c.bioch.virginia.edu/fasta_www2/fasta_list2.shtml`.

[62] U.S. Department of Energy Human Genome Program. Genomics and its impact on science and society: The Human Genome Project and beyond. `http://www.ornl.gov/TechResources/Human_Genome/publicat/primer2001/index.html`.

[63] Rüdiger Valk. Self-modifying nets, a natural extension of Petri nets. In *ICALP*, pages 464–476, 1978.

[64] Rüdiger Valk. Object Petri nets: Using the nets-within-nets paradigm. In *Lectures on Concurrency and Petri Nets*, pages 819–848, 2003.

[65] Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Trans. Comput. Logic*, 9(1):3, 2007.

[66] Wil M. P. van der Aalst. Verification of workflow nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426, London, UK, 1997. Springer-Verlag.

[67] Stijn Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Informatica*, 41(6):367–381, 2005.

[68] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[69] Philip Wadler. XQuery: a typed functional language for querying XML.

[70] L. Wong. Querying nested collections. PhD thesis, U.Pennsylvania, 1994.

[71] L. Wong. The collection programming language reference manual. `http://sdmc.iss.nus.sg/kleisli/psZ/cpl-defn.ps`, 1995.

[72] Extensible markup language (XML) 1.0 (fourth edition). `http://www.w3.org/TR/REC-xml`.

[73] XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery`.